Table des matières

Avant-Propos

Ce cours est destiné à des étudiants de 2e année de licence en informatique. Il est souhaitable pour le suivre dans de bonnes conditions d'avoir quelques prérequis en algèbre, en structure des ordinateurs et éventuellement en programmation structurée (langage C). Seuls de brefs rappels seront faits lorsque cela sera nécessaire.

D'autre part, pour diverses raisons, ce cours prend pour support le jeu d'instructions des processeurs de la famille *80x86* et la syntaxe assembleur *AT&T*. Toutefois, l'objectif de ce cours est seulement de permettre à l'étudiant d'acquérir les concepts fondamentaux de la programmation en assembleur. Le cours est donc loin d'être exhaustif, mais après avoir suivi ce cours, l'apprentissage d'autres syntaxes, d'autres instructions et éventuellement d'autres jeux d'instructions devrait être trivial.

1. Introduction

Tous les langages de programmation dits « évolués » (« structurés », « orientés objet », etc.), qu'ils soient compilés ou interprétés doivent d'une manière ou d'une autre être « traduits » en langage machine avant de pouvoir être exécutés par un ordinateur.

Or, s'il est indiscutable que ces langages ont un grand nombre d'atouts pour le développement de logiciels complexes, en termes de conception, de lisibilité, de portabilité et de maintenance, le revers de la médaille est qu'ils masquent fortement sinon totalement, les limitations de la machine.

Ne pas avoir conscience de ces limitations peut pourtant avoir des conséquences fâcheuses :

- en terme de performance, dans le cas d'une sous-exploitation des capacités du processeur[1].

1 voir, par exemple, l'apparition de processeurs 64 bits et des *Single Instruction Multiple Data* (dont font partie les jeux d'instructions *MMX*, *SSE* et *3D now!)* dans les dernières versions des processeurs Intel et AMD. Les SIMD permettent de réaliser jusqu'à 16 opérations arithmétiques en parallèle. Rares sont les compilateurs exploitent ces instructions, encore plus rares sont les programmes qui en tirent profit. La plupart des ordinateurs personnels présents sur le marché ont un

- en terme de fiabilité, dans le cas d'une surestimation des capacités (programmes qui dysfonctionnent ou qui doivent être entièrement réécrits dès que le cadre d'utilisation s'élargit).

Or, s'il est presque impossible d'écrire un programme conséquent directement en langage machine, il est tout à fait possible de l'écrire en « assembleur », qui n'en est qu'une version légèrement « humanisée ». Connaître les rudiments de la programmation en assembleur peut donc donner des atouts considérables pour la compréhension et la maîtrise de tous les autres langages de programmation.

1.1. Petits rappels sur la structure des ordinateurs

1.1.1. Contenu des cases mémoires

La mémoire d'un ordinateur est découpée en particules élémentaires appelées bits (pour binary units). Chacune de ces particules peut prendre seulement 2 valeurs possibles (la valeur est dite *binaire*).

La valeur d'une particule élémentaire peut avoir une quantité indénombrable d'interprétations différentes : logique (ex : vrai ou faux), numérique (0 ou 1), couleur (ex. : blanc ou noir), géographique (ex. : gauche ou droite), etc. Dans la suite de ce cours, nous allons utiliser le plus souvent une interprétation numérique entière. Notons qu'une interprétation numérique entière peut être reliée à n'importe quel type d'interprétation.

Le nombre d'interprétations différentes d'un ensemble de bits est infini. Cependant, dans une interprétation donnée, le nombre d'éléments distincts que cet ensemble de bits peut représenter est lui déterminé. Par exemple, un ensemble non ordonné de *n* bits peut représenter seulement *n+1* valeurs différentes. Par exemple si l'on dispose de seulement 2 bits, nous aurons seulement 3 valeurs possibles : 00, 10 et 11 (01 étant équivalent à 10 en l'absence de notion d'ordre).

En revanche, si nous considérons non plus des ensembles de bits, mais des suites de bits, alors la capacité de représentation est considérablement plus

processeur 64 bits, mais l'immense majorité est vendue avec un système d'exploitation 32 bits !

importante. C'est ainsi qu'est structurée la mémoire d'un ordinateur : des suites de cases mémoires, qui sont elles-mêmes des suites de bits.

Le nombre de bits qui composent une case mémoire détermine sa capacité de représentation. Pour bien comprendre la relation entre nombre de bits et capacité de représentation, nous pouvons prendre comme exemple la formation des nombres entiers naturels en base décimale : un chiffre décimal peut représenter 10 valeurs différentes et une suite (c.-à-d. un nombre) de 2 chiffres décimaux peut représenter 100 valeurs différentes. Lorsque nos particules élémentaires peuvent prendre les valeurs entières de 0 à 9, la capacité de représentation d'une suite de *x* particules se calcule simplement :

a) 1 particule = 10 valeurs possibles (de 0 à 9)
b) 2 particules = 100 valeurs possibles (de 0 à 99)

c) x particules = 10^x valeurs possibles (de 0 à 10^x-1)

Mais puisque nous sommes ici dans une base binaire, nous avons :
a) 1 particule = 2 valeurs possibles (de 0 à 1)
b) 2 particules = 4 valeurs possibles (nombres binaires 00, 01, 10 et 11)

c) x particules = 2^x valeurs possibles (de 0 à 2^x-1)

La mémoire étant une suite de bits, la capacité de représentation n'est dépendante que de la taille de la mémoire disponible : pour représenter un objet qui peut prendre X valeurs différentes, il nous faudra utiliser

nécessairement un nombre Y de bits tel que $X \leq 2^Y$. Pour représenter Z objets pouvant prendre chacun X valeurs différentes, il nous faudra utiliser

nécessairement un nombre Y de bits tel que $X \leq 2^Y Z$, etc. Nous pouvons voir ici qu'à moins de disposer d'une capacité mémoire infinie, il nous sera **impossible** de représenter complètement l'espace des nombres entiers, rationnels, réels, complexes, etc.
En fait, quel que soit le langage de programmation utilisé, le programmeur devra toujours représenter les ensembles infinis et/ou indénombrables du monde réel par des ensembles finis et dénombrables. Les erreurs de programmation issues d'un oubli de cet axiome sont très fréquentes. Elles peuvent ne se révéler que dans des situations exceptionnelles, mais leurs conséquences peuvent être très importantes.

Pour des raisons de visibilité et pour faciliter la représentation des données, nous utiliserons assez souvent la base hexadécimale (base 16) pour représenter les valeurs. Dans cette base, les chiffres sont (par valeur croissante) : **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F**. Le gros avantage de cette base par rapport à la base décimale est que chaque nombre binaire de 4 bits peut être représenté par un seul chiffre hexadécimal.

La valeur d'un nombre peut être calculée comme étant la somme du produit de chaque chiffre par une puissance de la base. Par exemple, la valeur du nombre

de 3 chiffres $C_2C_1C_0$ exprimé en base b , sera $C_2 * b^2 + C_1 * b^1 + C_0 * b^0$.

Chaque chiffre a donc un poids différent dans la somme. Par convention, on dira que les chiffres les plus à gauche sont de poids fort et les chiffres les plus à droite sont de poids faible.

En parlant d'une partie d'un nombre, on parlera de partie haute pour la partie la plus à gauche et de partie basse pour la partie la plus à droite. Par exemple, dans la base hexadécimale, pour un nombre de valeur : **F9E8D7C6**, nous appellerons partie haute de 8 bits la valeur **F9**, partie haute de 16 bits la valeur **F9E8**, partie haute de 24 bits, la valeur **F9E8D7**, partie basse de 8 bits la valeur **C6**, partie basse de 16 bits la valeur **D7C6** et partie basse de 24 bits la valeur **E8D7C6**.

On utilisera ainsi les termes *poids fort*, *poids faible*, *partie haute*, *partie basse* pour désigner les sous-parties du contenu d'une case mémoire. Les tailles de cases mémoires ou de partie de case mémoire les plus utilisées sont :
— L'octet : c'est une case mémoire de 8 bits
— Le mot de 16 bits : une suite de 2 octets
— Le mot de 24 bits : une suite de 3 octets
— Le mot de 32 bits : une suite de 4 octets
— Le mot de 64 bits : une suite de 8 octets

1.1.2. Adresse des cases mémoires

Il faut maintenant pouvoir désigner une particule élémentaire précise parmi toutes celles qui sont disponibles. Une première idée est de donner à chacune une adresse unique. Bien entendu, cette adresse doit être elle-même représentée sous la forme d'une suite de bits. Or, comme nous l'avons vu plus

haut, pour pouvoir donner X adresses différentes à X bits différents, nous avons besoin d'un espace mémoire de Y bits, tels que $X \leq 2^Y$.

Par exemple, pour pouvoir donner 4 adresses différentes à 4 bits différents, il nous faudra 2 bits pour stocker les adresses. Plus généralement, l'espace mémoire nécessaire pour stocker les adresses est inversement proportionnel à la taille minimale des cases mémoires adressables.

Dans la plupart des architectures de processeurs, cette taille minimale sera de 8 bits : chaque octet de la mémoire possède donc une adresse (physique) unique.

On aura donc besoin de 1 octet (8 bits) pour pouvoir faire référence à 256 (2^8) adresses différentes d'octets.

Avec 2 octets (16 bits) on pourra accéder à 65536 (2^{16}) octets (soit 64 kilo-octets), avec 4 octets (32 bits) on pourra accéder à 4 giga-octets (1 kilo-octet= 2^{10} octets, 1 mega-octet = 2^{10} kilo-octets et 1 giga-octet = 2^{10} mega-octet).

Les microprocesseurs peuvent manipuler directement des adresses sur 64 bits et peuvent donc gérer directement un espace adressable de 2^{64} octets, soit 16 exa-octet (16384 péta-octet, soit environ 17 milliards de giga-octets).

1.2. Qu'est-ce qu'un programme en langage machine ?

- Un programme en langage machine est une suite d'instructions-machine.
- Une instruction-machine est une suite de bits qui contient toutes les informations nécessaires à l'exécution de l'instruction.

D'un point de vue logique, c'est-à-dire du point de vue du programmeur, le microprocesseur va exécuter chaque instruction les unes après les autres. De fait, chaque instruction possède une adresse. Idéalement, cette adresse pourrait être simplement le rang de l'instruction dans la suite, c'est à dire : 1re instruction du programme, 2e instruction du programme, etc. Or, les instructions sont stockées dans la mémoire de l'ordinateur. Le microprocesseur y accède par blocs de 2^3x2^n bits, le plus souvent 8 bits (1 octet), 16 bits (un mot), 32 bits (un mot long) et plus récemment 64 bits. Plus rarement, on manipulera des données de 24 bits (par exemple pour le traitement d'images)

ou des données de 80 bits (nombres flottants étendus).

Les instructions sont de taille variable. En langage machine, l'adresse d'une instruction est donc l'*adresse mémoire* ou commence cette instruction.

Voici par exemple, un petit programme en langage machine totalement factice, écrit en binaire avec une mémoire découpée en octets :

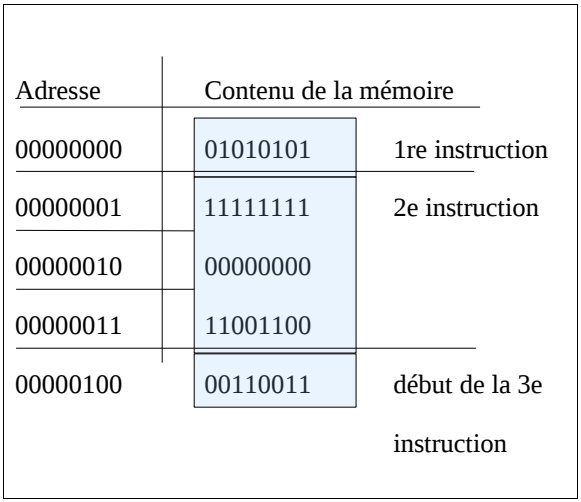


Figure 1 : programme en langage machine (factice)

1.3. Qu'est-ce qu'un programme en assembleur ?

L'assembleur permet simplement de rendre le langage machine un peu plus lisible :

- Chaque ligne d'assembleur contient une seule instruction, l'adresse d'une instruction est essentiellement constituée par son numéro de ligne.
- Le programmeur peut donner des noms aux adresses importantes pour lui.
- Les instructions s'écrivent sous la forme « *mnémonique suite_d'opérandes* » où la mnémonique est un mot qui rappelle le rôle de

l'instruction. Par exemple, la mnémonique *MOV* indique une instruction de transfert de données (en anglais *déplacer* se dit *move*). Les opérandes eux aussi peuvent avoir des noms définis par le programmeur. Par exemple, l'instruction assembleur « *MOV ORIGINE, DESTINATION* » va copier la donnée qui se trouve à l'adresse *ORIGINE* dans le contenu de l'adresse *DESTINATION*.

Un exemple de programme assembleur est donné en figure 2. La première colonne contient les *adresses symboliques* (appelées aussi *étiquettes*, par exemple *nb1*). Le programme chargé de l'assemblage (c'est-à-dire la translation assembleur vers langage machine) fera la transformation en adresses binaires.

Figure 2 : Programme en assembleur (syntaxe AT&T[2])

La deuxième colonne contient les *mnémoniques* (ex.: *movb*) et les *directives*

```
       .data         #(1) zone de données

nb1:   .byte 1       #(2) premier octet :
                     # valeur 1 en base décimale
res:   .byte 0       #(3) troisième octet :
                     #valeur 0 en base décimale

       .text         #(4) zone d'instructions

Addition :
              #(5) copier la valeur contenue dans l'octet
              # d'adresse nb1 dans le registre al
       movb   nb1,   %al
              #(6) ajouter la constante entière de
              # valeur 5 en base décimale au registre al
       addb   $5,    %al
              #(7) copier la valeur contenue dans le
              # registre al (1 octet) à l'adresse res
       movb   %al,   res
```

2 Les 2 principales syntaxes de l'assembleur sont la syntaxe *Intel* et la syntaxe *AT&T*. Dans ce cours, nous utilisons la syntaxe *AT&T*. C'est celle qui est utilisée par le compilateur-assembleur *GCC/GAS* de *GNU* (*Free Software Foundation)*. Ceci nous

(ex: *.data*). Les directives sont des instructions qui s'adressent au programme d'assemblage et qui lui permettent de produire un code exécutable compatible avec le système d'exploitation ciblé. Par exemple *.data* demande la création d'une zone de donnée, *.byte* demande la création d'un ou plusieurs espaces mémoire de 1 octet qui contiendront des nombres entiers préinitialisés.

Les colonnes suivantes contiennent les opérandes séparés les uns des autres, le cas échéant, par des virgules. Enfin, la dernière colonne contient généralement les commentaires du programme; ils décrivent instruction par instruction les opérations qui seront exécutées par le microprocesseur. L'exemple en figure 2 nous permet d'introduire plusieurs notions :

1.3.1 Type d'un opérande

Nous voyons dans l'exemple ci-dessus, deux types d'opérandes :

- nb1 et res sont des *étiquettes*. Elles désignent de façon symbolique des *adresses mémoire*.

- $5 est une constante entière, en base décimale. On pourra écrire par exemple $0**b**010101 pour exprimer la valeur d'une constante en base **b**inaire et $0**x**9ABC01 pour la base he**x**adécimale (base 16).

- al est un *registre*, pour distinguer un registre d'une étiquette, nous les faisons précéder par un signe « % »

Les registres sont des emplacements de mémoire internes au processeur. Les opérations arithmétiques ne peuvent être réalisées directement sur des emplacements en mémoire externe. Dans l'exemple précédent, il aurait été plus simple d'écrire une seule instruction « addb nb1, 5, res » réalisant la même opération. Malheureusement, les microprocesseurs de la famille 80x86 ne permettent pas ce type d'opérations à trois opérandes distinctes.

1.3.2. Taille d'un opérande ou d'une opération

La lettre « b » qui termine les mnémoniques *mov* et *add* indique la taille en bits de l'opération à réaliser.

---

permet de réaliser les travaux pratiques aussi bien sur GNU/Linux que sur Windows.

- « **b** » indique qu'il s'agit d'une opération sur 8 bits. Les opérandes indiquent donc des emplacements d'octets (en anglais : ***b****yte*).

- « **w** » indique une opération sur 16 bits. Les emplacements contiennent donc des mots (en anglais : ***w****ord*).

- « **l** » indique une opération sur des cases mémoires de 32 bits, soit des mots longs (en anglais ***l****ong words)*. On peut aussi utiliser le suffixe « **d** » pour ***d****ouble-word*.

- « **q** » indique une opération sur des cases mémoires de 64 bits, soit des mots quadruples (en anglais ***q****uad-word*).

1.3.3. Sens des opérations

Avec la syntaxe AT&T, lorsqu'il y a 2 opérandes, la première représente la *source* et la deuxième représente la *destination*. Voyons maintenant l'évolution de la mémoire interne et externe au fur et à mesure que notre programme est exécuté ligne après ligne :

ligne /instruction	***contenus***		
	nb1	***%al***	***res***
1/ « .data »	**?**	**?**	**?**
2/ nb1: .byte 1	**1**	**?**	**?**
3/ res: .byte 0	**1**	**?**	**0**
5/ movb nb1, %al	**1**	**1**	**0**
6/ addb $5, %al	**1**	**6**	**0**
7/ movb %al, res	**1**	**6**	**6**

Le code assembleur que nous venons d'écrire est donc susceptible d'être produit par la compilation d'une instruction simple en langage C ou C++ : « short int nb1=1, res=0; res=nb1+5; »

En assembleur, il n'y a pas de notion de « variable », les notions d'adresse symbolique, de registre, de taille, etc. sont très primitives comparées aux notions de variables (typées, structurées, de portée définie, etc.) telles qu'on peut les trouver dans les langages évolués.

2. Arithmétique entière

2.1. Un mot sur les registres généraux et leur capacité

En réalité, le registre ***al***, que nous avons vu rapidement en section 1.3, n'est que la partie basse (*accumulator low* en anglais) de 8 bits d'un registre de 16 bits nommé *ax* (*accumulator extended*). La partie haute de 8 bits de ce registre ***ax*** se nomme ***ah*** (pour *accumulator high)*. La version 32 bits de ce registre se nomme **eax** (*extended accumulateur extended*, bits numérotés de 0 à 31) et la version 64 bits du même registre se nomme **rax**.

Autrement dit, e*ax* représente les 32 bits de poids faible de rax (bits 0 à 31), *ax* représente les 16 bits de poids faible de eax (bits 0 à 15), *al* représente ses 8 bits de poids faible (bits 0 à7) et *ah* représente les bits de numérotés 8 à 15 dans la figure suivante :

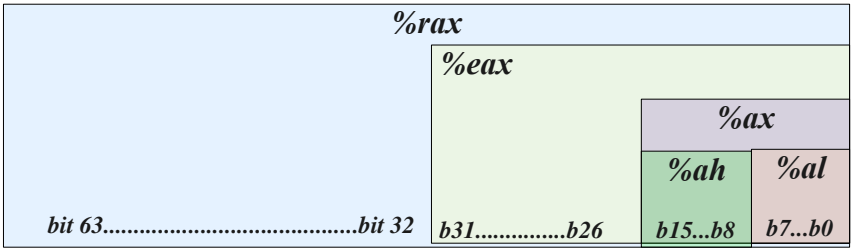


Figure 2 : Le registre rax (64 bits)

Les processeurs récents de la famille 80x86 (AMD® et Intel®) possèdent 3 autres registres de 64 bits similaires à *rax : rbx*, *rcx* et *rdx*. Ils sont eux aussi décomposables en registres de 8, 16 et 32 bits : *ebx*, *ecx*, *edx*, *bx*, *cx*, *dx*, *bh*, *ch*, *dh*, *bl*, *cl* et *dl*.

Il est important de bien comprendre la structure imbriquée des registres et sous-registres : seul *ah* et *al* sont indépendants. Dans les autres cas, une modification d'un des registres a des répercussions sur les autres registres.

On peut légitimement se demander pourquoi les registres ont une structure aussi complexe. En fait, ce choix technologique découle simplement de

nécessités économiques : Les premiers processeurs de la famille 80x86[3] avaient un bus de données de 8 bits et un bus d'adresse de 16 bits. Mais à chaque évolution de la taille des bus de données et d'adresses, il fallait garantir que les logiciels écrits pour la génération de processeurs précédente fonctionnent parfaitement avec les nouveaux processeurs. Sans cette garantie, il aurait été nécessaire de recompiler l'ensemble du parc logiciel. De ceci a dérivé une contrainte simple : chaque nouveau jeu d'instructions et de registres doit être un sur-ensemble des anciens jeux d'instructions et de registres.

2.2. Entiers signés et non signés

Revenons sur les questions de capacité de représentation des données. Si l'on considère les entiers naturels, la valeur la plus basse est bien entendu 0 et la valeur maximale est donné par la formule 2^n-1, n étant le nombre de bits utilisés pour le stockage. Autrement dit, un octet (8 bits) peut contenir des valeurs entières allant de 0 à 255, un mot (16 bits) des valeurs allant de 0 à 65.535, un mot long (32 bits) des valeurs allant de 0 à 4.294.967.295 et un mot quadruple (64 bits) des valeurs allant de 0 à 18.446.744.073.709.551.615.

Qu'en est-il pour les entiers relatifs ? Il est évident que d'une façon ou d'une autre, nous devrons utiliser 1 bit pour stocker le signe du nombre. Nous pourrions naturellement utiliser 1 bit pour stocker le signe et tous les autres bits pour stocker la valeur absolue. Mais il est aussi évident que cela nous obligerait à modifier l'algorithme que nous utilisons pour l'addition et la soustraction des entiers naturels. Par exemple, sur un octet, 0b1 – 0b11 devrait donner comme résultat : 0b10000001 alors que l'algorithme de soustraction pour les entiers naturels nous donne : 0b11111111.

Nous allons plutôt rechercher un type de représentation qui modifie le moins possible cet algorithme additif. La première constatation est que si l'on choisi le bit de poids fort comme bit de signe avec sa valeur 0 comme indication d'un nombre positif, hormis la question des capacités, l'addition et la soustraction fonctionneront comme avant.

3 Le 8086, présenté par Intel en1978 était un processeur 16 bits. Le 80386, présenté par Intel en 1985 permit de passer à 32 bits. Et il a fallu attendre les années 2000 pour voir apparaître les premiers 64 bits de la famille x86 par AMD.

Par exemple, l'addition des deux entiers relatifs 64 et 63 (résultat 127), s'écrira en binaire :

	0	1	0	0	0	0	0	0
+	0	0	1	1	1	1	1	1
=	0	1	1	1	1	1	1	1

En ce qui concerne les nombres négatifs, regardons ce qu'il se passe si l'on représente les nombre sur 3 bits : La valeur positive maximale est 3 (0b011), en soustrayant successivement 1 à cette valeur, nous obtenons :

Binaire	***Décimal***
011	3
010	2
001	1
000	0
111	-1
110	-2
101	-3
100	-4

En théorie, pour l'opération 0b0-0b1, notre algorithme « naturel » de

soustraction par propagation des retenues donne un nombre binaire composé d'une infinité de bits à 1. Mais seuls les 3 bits de poids faible du résultat peuvent être stockés.

Ce mode de représentation des nombres négatifs se nomme le *complément à 2*. Pour obtenir l'opposé d'un nombre, il suffit d'inverser valeur chacun des bits de ce nombre et de lui ajouter 1. Par exemple, l'opposé de 0b011 (3) sur 3 bits est 0b100+0b1 = 0b101 (-3). Notez que si on ajoute *naturellement* un nombre à son opposé, on obtient bien la valeur 0 sur 3 bits.

Lorsqu'il s'agit d'entiers signés, les valeurs seront donc comprises entre -2^{n-1} et $2^{n-1}-1$, n étant le nombre de bits utilisés pour le stockage. Notez que l'opposé de la valeur minimale (-2^{n-1}) ne donne pas un résultat valide ($2^{n-1}>2^{n-1}-1$). Par exemple, l'opposé de 0b100 (-4) est 0b100.

Ici, on voit bien apparaître la différence entre les ensembles des entiers naturels ($\mathbb{N}$), des entiers relatifs ($\mathbb{Z}$), des entiers non signés sur x bits ($\mathbb{N}_{x)}$ et entiers signés sur x bits ($\mathbb{Z}_x$). On a les relations suivantes :$\mathbb{N}\subset\mathbb{Z}$, $\mathbb{Z}_x\subset\mathbb{Z}$ et $\mathbb{N}_x\subset\mathbb{N}$, mais il faut faire attention, car $\mathbb{N}_x\not\subset\mathbb{Z}_x$, par contre $\mathbb{N}_x\subset\mathbb{Z}_{x+1}$.

Et bien entendu, toutes les opérations arithmétiques sont susceptibles d'amener nos valeurs à dépasser les bornes minimales ou maximales de nos entiers...

2.3. Débordements

Comment le programmeur peut-il savoir si le calcul a donné un résultat valide ? En effet, si l'on ajoute 1 à 255 sur un octet, pour le microprocesseur, le résultat (invalide) est 0. De même, toujours sur un octet, -128 — 1 donne 127 (0b10000000 – 0b1 = 0b01111111).

En plus des registres généraux rax, rbx, rcx, rdx, le microprocesseur possède un registre spécial : le registre d'état *RFLAGS* (FLAG signifie « Drapeau » en anglais). Chacun des 64 bits (drapeaux) de ce registre indique une caractéristique binaire de l'état du processeur. Mais seule une petite partie de ces bits concernent les opérations arithmétiques. Après exécution d'une instruction arithmétique, le bit :

- ZF (*Zero Flag, bit n°6 de RFLAGS*) indique si le résultat est nul (ZF=1) ou non nul (ZF=0).

- SF (*Sign Flag, bit n°7 de RFLAGS*) indique si le résultat est positif (SF=0) ou négatif (SF=1).

- PF (*Parity Flag, bit n°2 de RFLAGS*) indique que le résultat est pair (PF=1) ou impair (PF=0).

- CF (*Carry Flag, bit n°0 de RFLAGS*) indique une *retenue* (CF=1) sur les entiers non signés.

- OF (*Overflow Flag, bit n°11 de RFLAGS*), indique un *débordement* (OF=1) sur les entiers signés.

Lors d'une opération sur *n* bits, le drapeau CF contiendra le bit numéro *n* (ou *n-1*ème bit) du résultat. Par exemple, sur 8 bits :

- 0b11111111 + 0b1 = 0b00000000 avec CF = 1

- 0b00000000 – 0b1 = 0b11111111 avec CF = 1

- 0b11111110 + 0b1 = 0b11111111 avec CF = 0

On voit donc aisément, que si pour le programmeur, les opérandes sont des entiers non signés, le résultat est valide si et seulement si CF =0 et il est invalide si et seulement si CF = 1.

Si pour le programmeur, les opérandes sont des entiers signés, c'est le drapeau OF qui indiquera la validité du résultat. Par exemple, sur 8 bits (non-signés : $\mathbb{N}_8$, signés : $\mathbb{Z}_8$) :

- 0b11111111 + 0b1 = 0b00000000 avec CF=1 et OF=0 ,

  (255+1) est invalide dans $\mathbb{N}_8$, mais (-1+1) est valide dans $\mathbb{N}_8$.

- 0b00000000 – 0b1 = 0b11111111 avec CF=1, mais OF=0,

  (0-1) est invalide dans $\mathbb{N}_8$, mais (0-1) est valide dans $\mathbb{N}_8$.

- 0b10000000 + 0b1 = 0b10000001 avec CF=0 et OF=0,

  (128 + 1) est valide dans $\mathbb{Z}_8$ et (-128 + 1) est valide dans $\mathbb{Z}_8$.

- 0b10000000 – 0b1 = 0b01111111 avec CF=0, mais OF =1,

  (128–1) est valide dans $\mathbb{Z}_8$, mais (-128–1) est invalide dans $\mathbb{Z}_8$.

- 0b01111111 + 0b1 = 0b10000000 avec CF=0, mais OF =1,

  (127+1) est valide dans $\mathbb{Z}_8$, mais (127+1) est invalide dans $\mathbb{Z}_8$.

3. Instructions de branchement

Nous sommes maintenant en mesure de programmer des formules arithmétiques simples. Laissons pour le moment les instructions de calcul et la représentation des nombres pour nous pencher sur ce qui va nous permettre d'aborder des algorithmes un peu plus complexes : les instructions de branchement.

3.1. Le registre RIP (Re-Extended Instruction Pointer)

Nous avons vu que les instructions – de taille variable — étaient exécutées par le processeur les unes après les autres. Pour cela, le processeur dispose d'un registre spécial, nommé *RIP* d'une capacité de 64 bits (nommé *EIP* pour sa partie basse de 32 bits et *IP* pour sa partie basse de 16 bits[4]) qui contient l'adresse de l'instruction courante à exécuter. La valeur contenue dans ce registre est automatiquement augmentée lors de l'exécution d'une instruction afin que le registre pointe sur l'instruction suivante (c'est-à-dire la ligne suivante dans un programme assembleur écrit proprement). Nous allons maintenant étudier des instructions qui modifient directement le contenu du registre *RIP*.

3.2. Branchement inconditionnel : instruction jmp (de « jump » : sauter)

L'instruction « **jmp <adresse>** » permet de remplacer le contenu de *RIP* par une adresse symbolique ou une constante. Par conséquent, l'instruction exécutée après l'instruction *jmp* n'est plus celle qui se trouve sur la ligne suivante, mais celle située à l'adresse donnée en opérande de la mnémonique *jmp*.

Par exemple, dans le programme ci-dessous, l'instruction « addl $2, %eax » ne sera jamais exécutée et à la fin de l'exécution eax et ebx contiendront la valeur 2 et non la valeur 4.

```
debut:      movl $2, %eax     # 2 -> eax
```

4 Les anciens processeurs de la famille 80x86 avaient un bus d'adresse de 16 bits. La compatibilité ascendante a été respectée, mais le registre IP est n'est utilisable que dans des conditions très particulières que nous n'aborderons pas dans ce cours

```
            jmp suite          # on saute l'
                               # instruction suivante
            addl $2, %eax      # cette instruction
                               # n'est pas exécutée
suite:      movl %eax, %ebx    # eax -> %ebx
```

Le saut peut avoir avoir lieu *en avant* ou *en arrière* ainsi la ligne suivante est ce qu'on appelle une boucle infinie :

```
debut:      jmp debut    # mets l'adresse debut
                         # dans le registre RIP
```

3.3. Branchements conditionnels

Les instructions de saut conditionnel testent un ou plusieurs drapeaux du registre d'état et en fonction de leur valeur, effectuent le branchement ou passent à l'instruction suivante. La syntaxe de ces instructions est la même que pour la mnémonique *jmp :* il y a un seul opérande ; c'est l'adresse ou a lieu de branchement si la condition du branchement est remplie. Par exemple avec les drapeaux ZF, *CF et OF* que nous avons vu, nous pouvons utiliser les instructions suivantes :

- jz : « jump if zero », c'est à dire « saut si résultat nul». Le branchement est effectué si ZF =1.
- jnz : « jump not zero », c'est-à-dire « saut si résultat non nul ». Le branchement est effectué si ZF =0.
- jc : « jump if carry », c'est-à-dire « saut si retenue ». Le branchement est effectué si CF =1.
- *jnc :* « jump if not carry », c'est-à-dire « saut si pas de retenue ». Le branchement est effectué si *CF =0.*
- *jo :* « jump if overflow », c'est-à-dire « saut si débordement ». Le branchement est effectué si *OF =1.*
- *jno :* « jump if not overflow », c'est-à-dire « saut si pas de débordement ». Le branchement est effectué si *OF =0.*

Il est aisé de voir que ces 4 dernières instructions seront très utiles pour diriger le programme vers un traitement particulier lorsqu'un calcul rend un résultat invalide dans les entiers naturels ou relatifs. Par exemple :

```
Init:       movw $0, %cx  # 0 -> cx
Boucle:     addw $1, %cx  # cx+1 -> cx
            jnc Boucle    # boucle tant que cx+1
                          # est valide (de 1 à 65535)
```

Ce programme est une boucle qui va incrémenter le registre *cx* de 1 à chaque itération. On sortira de la boucle lorsque CF sera égal à 1, c'est à dire lorsque l'incrémentation de cx provoquera une retenue d'entier non signé. Notons que toutes les structures de contrôle des langages évolués (if-then-else, while, for, repeat-until, do, case, etc.) peuvent être réalisées avec les seules instructions que vous avez vues jusqu'à maintenant. Par exemple, l'instruction Pascal « if a>b then c:=a else c:=b; », a, b et c étant des variables globales entières non signées 64 bits, pourrait s'écrire :

```
maximum:    movq var_a, %rax  # a -> rax
            movq var_b, %rbx  # b -> rbx
            subq %rax, %rbx   # rbx-rax -> rbx
            jc amax           # CF=1 => b-a<0 => a>b
bmax:       movq var_b, %rax  # CF=0 => b-a>=0 =>
                              # a <= b, on copie
                              # b dans rax
amax:       movq %rax, var_c  # max(a,b) -> c
```

4. Structure des données : pointeurs, tableaux, matrices, etc.

4.1. Registres « pointeurs »

Toutes les structures de données avancées reposent sur le concept de pointeur. Un pointeur est une case mémoire qui contient l'adresse d'une autre case mémoire. Comme on peut modifier le contenu du pointeur, on pourra accéder

à différentes zones de la mémoire à partir d'une position initiale. Toutes les structures de données complexes reposent sur ce principe : tableaux, structures, matrices, etc.

Le seul registre « pointeur » que nous avons vu jusqu'à maintenant est r*ip*. Il a un rôle très particulier et il n'est pas facilement utilisable pour d'autres tâches que celle de pointer sur l'instruction courante. Les registres généraux r*ax*, r*bx*, r*cx* et r*dx* peuvent servir comme pointeurs. Deux autres registres r*si* (re-extended source index) et r*di* (re-extended destination index) sont souvent utilisés respectivement comme pointeur indiquant la source et comme pointeur indiquant la destination dans des opérations de copie de zones mémoires. Enfin, nous verrons plus loin le pointeur r*sp* (re-extended stack pointer) et r*bp* (re-extended base pointer) qui seront utilisés pour la gestion de la pile.

4.2. Mode d'adressage « indirect »

Nous avons vu :

1. L'adressage immédiat (constante), par exemple : $12, $0b010, $0x1ABC, etc.

2. L'adressage registre, par exemple : %rax, %rcx, etc.

3. L'adressage direct, par exemple : caseB, var_a, etc.

Pour manipuler des structures de données plus complexes, nous aurons besoin au minimum du **mode d'adressage indirect**, par exemple :

```
movq $Tableau, %rsi     # fait pointer %rsi sur
                        # le 1er élément du tableau
movw (%rsi), %ax        # Copie l'élément
                        # courant du tableau dans ax
addw $1, %ax            # Ajoute 1 à ax
movw %ax, (%rsi)        # Remplace l'élément
                        # courant du tableau par ax
addl $2, %rsi           # Ajoute 2 à rsi
                        # (passe à l'élément suivant)
```

Dans cet exemple, nous avons un tableau qui contient des valeurs de 16 bits (2 octets). Cet extrait de programme ajoute la valeur 1 au premier élément et fait pointer r*si* sur l'élément suivant. On comprend bien ici le rôle des parenthèses qui entourent le nom du registre : *%rsi* désigne le contenu du registre *rsi* alors que *(%rsi)* désigne le contenu de la case mémoire dont l'adresse est contenue dans *rsi*. **Ce concept est fondamental pour la compréhension du fonctionnement des structures de données dans tous les langages de programmation.**

Nous pouvons maintenant écrire presque tout algorithme avec les seuls concepts, registres et instructions que nous avons vu jusqu'à présent. Autrement dit, on peut concevoir un compilateur pour n'importe quel langage évolué qui produirait du code composé uniquement des instructions, registres et mode d'adressages expliqués ci-dessus. Nous aurions même pu nous passer de certaines instructions ou certains registres. Ce qui va suivre peut malgré tout permettre d'écrire des exécutables considérablement plus compacts et efficaces.

4.3 Mode d'adressage « indirect indexé »

Il est possible en utilisant ce mode d'adressage, d'effectuer un déplacement relatif par rapport au pointeur. En reprenant l'exemple de la section *4.2*, on pourrait copier directement le 2e élément du tableau dans le registre *bx* avec l'instruction :

```
movw 2(%rsi), %bx
```

Avec ce mode d'adressage, le processeur récupère l'adresse contenue dans *rsi*, ajoute à cette adresse la constante qui précède la parenthèse ouvrante et copie le contenu de l'adresse ainsi obtenue dans le registre *bx*. Il est important de noter que :

- **Le contenu de *rsi* reste inchangé.** si %rsi contient la valeur *$0x0102030405060708* avant l'exécution de « movw 2(%rsi), %bx », il contient toujours *$0x0102030405060708* après l'exécution de cette même instruction et non *$0x010203040506070A* bien que ce soit les 16 bits trouvés à cet emplacement qui sont copiés dans *%bx*.

- Le déplacement relatif peut être positif ou négatif, mais **il ne peut être**

qu'une constante. Par exemple, si *rsi* contient l'adresse du dernier élément du tableau, alors *-2(%rsi)* désignera l'avant-dernier élément de 16 bits du tableau. Par contre, le mode d'adressage *%bx(%rsi)* n'est pas autorisé puisque *%bx* n'est pas une constante.

Nous imaginons facilement comment ce mode d'adressage va se révéler utile pour traiter des données du type *record* (en pascal) ou *structure* (en C) :

```
movq $fiche, %rsi      # rsi pointe sur une fiche
movl (%rsi), %eax      # copie le numéro de
                       # téléphone (4 octets) dans eax
movb 4(%rsi), %bl      # copie l'âge de la personne
                       #(1 octet) dans bl
movw 5(%rsi), %cx      # copie l'année de naissance
```

rsi	***fiche***	***fiche+1***	***fiche+2***	***fiche+3***
$fiche	**0d490843500**			
	(%rsi) ☎	1(%rsi) ☎	2(%rsi) ☎	3(%rsi) ☎

fiche+4	***fiche+5***	***fiche+6***	***fiche+7***
0d24	**0d1980**		
4(%rsi) âge	5(%rsi) an. naiss.	6(%rsi) an. naiss.	7(%rsi)

4.4. Indexations complexes

Dans le même esprit, on pourra écrire *-5(%rsi, %rax)* pour désigner le contenu de la case mémoire pointée par *rsi+rax-5*. Comme nous avons vu ci-dessus, le calcul est réalisé par le processeur : *rsi* et *rax* restent inchangés lors d'une telle opération. Il faut noter que les deux registres doivent être de même taille. Ici %rsi est appelé « base » et %rax est appelé « index ». La taille des deux registres dépend donc du type de processeur (16, 32 ou 64) et du type de système d'exploitation : un système GNU/Linux ou Windows 32 bits ne permettra pas d'utiliser les caractéristiques 64 bits d'un processeur 64 bits.

Il existe un mode d'adressage encore plus puissant : *5(%rdi, %rbx, 8)* désigne ainsi le contenu de la mémoire dont l'adresse est calculée par : *rdi+(rbx*8)+5*. Reprenons l'exemple de la section 4.3. et supposons que *rsi* pointe sur la première fiche d'un tableau de fiche et que *rbx* contient le numéro de la fiche courante :

```
movb 4(%rsi, %rbx, 8), %al    # Fiche[rbx].age -> %al
```

Cette instruction aura pour effet de copier dans *al*, la valeur pointée par *rsi+(rbx*8)+4*, c'est-à-dire l'adresse du champ « âge » de la fiche numéro %*rbx*. Ici, la valeur 8 est appelée « scale factor ». Ce champ peut prendre uniquement les valeurs 1, 2, 4 ou 8 qui correspondent aux tailles octets, mot de 16 bits, mot de 32 bits et mot de 64 bits.

5. Comparaisons et autres branchements conditionnels

Nous avons vu ci-dessus comment réaliser une comparaison arithmétique en

utilisant la soustraction. Pour des raisons de lisibilité et d'efficacité, il est en fait préférable d'utiliser une instruction prévue spécialement pour ça : la mnémonique *cmp*.

***« cmp* opérande1, *opérande2* »** est une sorte de soustraction virtuelle. Plus précisément, les contenus des *deux opérandes* restent inchangés, mais le registre *RFLAGS* est modifié exactement comme lors de la soustraction **opérande2-opérande1**. Les drapeaux *OF*, *CF*, *ZF*, *SF* et *PF* donnent les caractéristiques du résultat.

Après une instruction de comparaison, nous pouvons bien sûr faire un branchement conditionnel en fonction des valeurs de *RFLAGS*. Pour plus de lisibilité, chaque mnémonique de branchement conditionnel possède plusieurs synonymes. Par exemple :

- Saut si CF = 1 : ***jc*** (jump[5] if carry[6]), ***jb*** (jump if below[7]), ***jnae*** (jump if not above[8] or equal[9])
- Saut si CF = 0 : ***jnc*** (jump if no carry), ***jae*** (jump if above or equal), ***jnb*** (jump if not below)
- Saut si ZF = 1 : ***jz*** (jump if zero), ***je*** (jump if equal)
- Saut si ZF = 0 : ***jnz*** (jump if not zero), ***jne*** (jump if not equal)
- Saut si CF=0 et ZF=0 : ***ja*** (jump if above), ***jnbe*** (jump if not below or equal).
- Saut si CF=1 ou ZF=1: ***jbe*** (jump if below or equal), ***jna*** (jump if not above)

Bien entendu, jb, jnae, jae, jnb, ja, jnbe, jbe et jna n'ont de sens que si l'instruction précédente est une comparaison d'entiers non signés (naturels). Si l'on veut comparer des entiers relatifs (signés), on utilisera :

5 en français : saute
6 en français : retenue
7 en français : en dessous
8 en français : au-dessus
9 en français : égal

- Saut si ((SF oux[10] OF) ou ZF) = 0 : ***jg*** (jump if greater[11]), ***jnle*** (jump if not less[12] or equal)
- Saut si ((SF oux OF) ou ZF) =1 : ***jle*** (jump if less or equal), ***jng*** (jump if not greater)
- Saut si (SF oux OF) =0 : ***jge*** (jump if greater or equal), ***jnl*** (jump if not less)
- Saut si (SF oux OF) = 1 : ***jl*** (jump if less), ***jnge*** (jump if not greater or equal)
- Saut si OF = 0 : ***jno***(jump if not overflow)
- Saut si OF = 1 : ***jo***(jump if overflow)
- Saut si SF = 0 : ***js*** (jump if sign : saut si résultat négatif)
- Saut si SF = 0 : ***jns*** (jump if not sign : saut si résultat positif).

Enfin, 4 instructions de saut conditionnel un peu particulières :

- Saut si PF =0 : ***jnp***, ***jpo*** (*not parity* ou *parity odd :* le résultat est impair)
- Saut si PF =1 : ***jp***, ***jpe*** (*parity* ou *parity even :* le résultat est pair).

Et comme le registre rcx est souvent utilisé comme compteur de boucle :

- Saut si %cx = 0 : ***jcxz*** (le registre cx contient 0)
- Saut si %ecx =0 : ***jecxz*** (le registre ecx contient 0)
- Saut si %ecx =0 : ***jrcxz*** (le registre rcx contient 0)

Par exemple, l'instruction Pascal « if a>b then c:=a else c:=b; », a, b et c étant

10 « ou exclusif » : (SF oux OF) = 1 lorsque soit SF=1 soit OF=1, mais est égal à 0 si SF=1 et OF=1
11 en français : plus grand
12 en français : plus petit

des variables globales entières non signées 32 bits, peut maintenant s'écrire :

```
maximum:    movl var_b, %eax  # var_b -> eax
            cmpl var_a, %eax  # compare var_a à eax
                              # (en fait : eax moins var_a)
            jae bmax          # jump if above or
                              # equal (vers bmax si b>=a)
amax:       movl var_a, %eax
                              # a > b, on copie a dans eax
bmax:       movl %eax, var_c  # max(a,b) -> c
```

Tableau récapitulatif des sauts conditionnels pour les entiers signés et non signés :

Condition après : `cmp a, b`	***Mnémonique***	***Condition de branchement***
a=b	JE/JZ	ZF=1
a≠b	JNE/JNZ	ZF=0

Tableau récapitulatif pour les entiers non signés :

Condition après : `cmp a, b`	***Mnémonique***	***Condition de branchement***
a>b	JA/JNBE	CF=0 et ZF=0
a≥b	JAE/JNB/JNC	CF=0
a<b	JB/JNAE/JC	CF=1
a≤b	JBE/JNA	CF=1 ou ZF=1

Tableau récapitulatif pour les entiers signés :

Condition après : cmp a, b	***Mnémonique***	***Condition de branchement***
a>b	JG/JNLE	((SF oux OF) ou ZF) = 0
a≥b	JGE/JNL	(SF oux OF) = 0
a<b	JL/JNGE	(SF oux OF) = 1
a≤b	JLE/JNG	((SF oux OF) ou ZF) = 1
a-b > min_Z[13]	JNO	OF = 0
a-b<min_Z	JO	OF =1
a-b<0	JNS	SF = 0
a-b>0	JS	SF =1

13 Pour rappel, min_z = -2^{n-1} , n étant le nombre de bits utilisés lors de la comparaison précédente.

6. Équivalents des structures algorithmiques avancées

Il n'y a pas, en assembleur de structures de boucles ou de choix multiples comme on peut les trouver dans les langages structurés de type C ou Pascal. Nous pouvons les réaliser, par exemple de la façon suivante (équivalent assembleur de structures réalisées en C) :

```
If(var_a  >  var_b)  {<instructions-alors>}  else
{<instructions-sinon>}

Si:         cmp   a,    b
            jae   Sinon
# b>=a  --> saut vers sinon
Alors:      <instructions-alors>
            jmp   FinIf
Sinon:      <instructions-sinon>
FinIf:
```

```
While(var_a > var_b) {<instructions>}

TantQue:    cmp   a,    b
            jae   FinTantQue
# b>=a  ---> saut vers FinTantQue
            <instructions>
            jmp TantQue
```

```
FinTantQue:
```

Do {<instructions>} While(a>b)

```
Faire:     <instructions>
           cmp   a,    b
           jb    Faire       # b<a --> boucle
FinFaire:
```

for(i=0; i<=10; i++) {<instructions>}

```
PourInit:         movq  $0,   %rax
PourTest:         cmpq  $10,  %rax
                  ja    FinPour     # i>10
                  <instructions>
                  addq $1,    %rax
                  jmp PourTest
FinPour:
```

```
switch (a) {
           case 'a':
                       <instructionsA>
                       break;
           case 'a':
                       <instructionsB>
                       break;
           default:
                       <instructionsC>
}

CasA:      cmpb  $'a', %al
           jne   CasB
           <instructionsA>
           jmp FinCas:
CasB:      cmpb  $'b', %al
           jne   Defaut
           <instructionsB>
           jmp FinCas:
Defaut:    <instructionsC>
FinCas:
```

7. Utilisation de la pile

Le CPU possède un registre spécial nommé %*rsp*[14] (pour re-*extended stack pointer*, soit pointeur de pile étendu). La pile est un tableau de cases mémoires contiguës. %*rsp* pointe en permanence sur le sommet de la pile, c'est-à-dire le dernier élément empilé. L'empilement d'une valeur (64 bits) se fait par l'instruction *pushq* et le dépilement par l'instruction *popq*. En fait, « push opérande » commence par **décrémenter** %*rsp* et remplace le contenu de la case pointée par *%rsp* par « opérande ». *pop* se contente d'**incrémenter** %*rsp*. L'incrémentation et la décrémentation se font par groupe de 8 octets, soit 64 bits. L'utilisation de la pile va se révéler très utile, par exemple pour simuler l'utilisation de « variables locales » ou de paramètres de procédures. D'autre part, nous verrons qu'il est possible d'accéder directement à des éléments de la pile sans passer par les instructions *push* et *pop*.

Si le processeur fonctionne en mode 32 bits (systèmes d'exploitation 32 bits), la pile est pointée par %esp, et les valeurs sont empilées (pushl) ou dépilées (popl) par groupes de 4 octets. Si le processeur fonctionne en mode 16 bits (systèmes d'exploitation 16 bits), la pile est pointée par %sp, et les valeurs sont empilées (pushw) ou dépilées (popw) par groupes de 2 octets.

8. Procédures

8.1 Les instructions call et ret

Les instructions *call* et *ret* permettent respectivement d'appeler et de sortir d'une procédure. L'opérande qui suit *call* est l'adresse symbolique de la procédure. Traditionnellement, les paramètres de la procédure sont passés par la pile.

call empile tout d'abord l'adresse de la prochaine instruction (le contenu de *%rip*) et remplace le contenu de *rip* par l'adresse donnée en opérande. *ret* dépile et place la valeur dépilée dans *rip*, ce qui provoque (sauf erreur de programmation) le retour à l'endroit où la procédure à été appelée (instruction qui suit le « *call adresse* »). Comme la pile n'est pas uniquement utilisée pour sauvegarder l'adresse de retour, mais aussi pour stocker les paramètres et les

14 Il existe une version 16 bits de ce registre (*sp*), mais nous ne pouvons l'utiliser que dans un mode particulier du processeur que nous ne verrons pas ici.

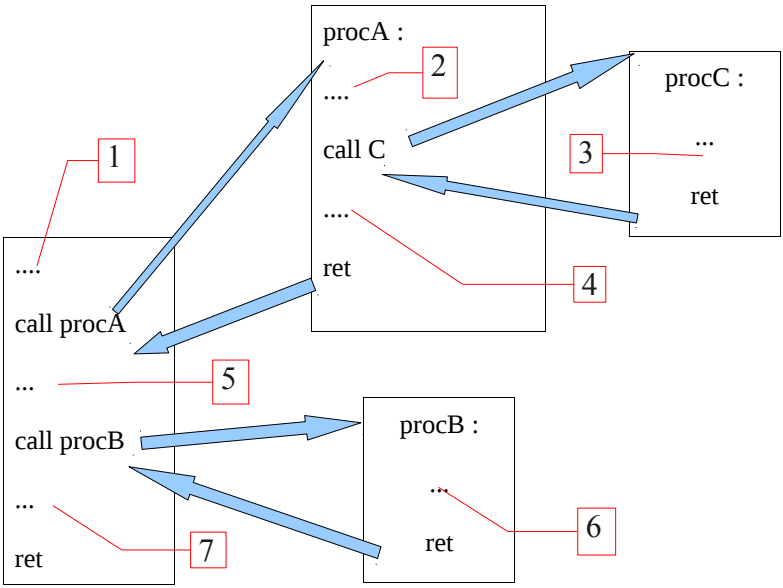
variables locales de la procédure, il est important de s'assurer que le haut de la pile contient bien l'adresse de retour avant l'exécution de *ret*.

ret peut avoir une opérande *n* de type « constante entière ». Dans ce cas, *ret* dépilera l'adresse de retour, puis *n* octets avant de retourner à la procédure appelante. Le registre r*bp* (*re-extended base pointer*) est souvent utilisé par le programmeur pour accéder aux différentes zones de la pile, il pointe généralement sur l'adresse de retour de la procédure.

Voici un exemple de procédure qui prend 2 entiers non signés comme paramètres (par la pile) et renvoie le maximum des 2 valeurs :

```
appel:
# Empilage des paramètres de la procédure Maximum
             push UnsignedQuadWord_a
             push UnsignedQuadWord_b
             call Maximum        # appel procédure Maximum
             popq   %rax         # résultat dans rax
# <<suite de la procédure appelante >>
             retq   # retour à la procédure appelante
Maximum:
             movq   %rsp, %rbp
                    # rbp pointe sur l'adresse de retour
             pushq %rax   # sauve ancienne valeur de rax
             pushq %rbx   # sauve ancienne valeur de rbx
             movq 8(%rbp), %rbx  # copie le paramètre
                                 # b dans rbx
             movq 16(%rbp), %rax # copie le paramètre
                                 # a dans rax
             cmpq %rax, %rbx     # compare le 1er et le 2e
                                 # paramètre
             jae    bmax   # Si b >= a alors saute à bmax
amax:        pushq %rax   # empile a
             jmp FinMax
bmax:        pushq %rbx   # empile b
FinMax:      popq %rax    # valeur max -> rax
             movq %rax, 8(%rbp)
# valeur max -> 1er paramètre
             popq %rbx  # récupère ancienne valeur de %rbx
             popq %rax  # récupère ancienne valeur de %rax
             retq $8    # dépile le 2e paramètre avant de
      # retourner à la procédure appelante
```

Voici un schéma qui illustre l'exécution de procédures imbriquées. Les chiffres représentent l'ordre d'exécution du code :



Et voici l'évolution de la pile lors de l'exécution schématisée ci-dessus :

1. {}
2. {adresse de 5}
3. {adresse de 5, adresse de 4}
4. {adresse de 5}
5. {}
6. {adresse de 7}
7. {}

8.2 Les interruptions et les exceptions

Il nous manque encore un dispositif pour permettre au processeur de réagir en temps réel à divers *évènements*, liés aux périphériques (appui d'une touche sur le clavier, clic ou mouvement de la souris, etc.) ou à l'état du processeur lui-même (division par zéro, accès à une zone mémoire protégée, etc.). Un simple *balayage* permanent de ces très nombreuses données gaspillerait un montant considérable de ressources.

Heureusement, tous les processeurs actuels possèdent un jeu d'instructions qui permettent une gestion événementielle de cette problématique. Par exemple, un périphérique, lorsqu'il est sollicité par l'utilisateur, envoie un signal (qu'on appelle interruption) au microprocesseur, qui peut alors interrompre l'exécution du programme courant, exécuter une routine du système d'exploitation et revenir au programme courant lorsque l'interruption a été traitée.

En réalité, n'importe quel programme peut générer des interruptions logicielles et des exceptions. Mais nous ne rentrerons pas ici dans des détails qui relèveraient plus d'un cours sur les systèmes d'exploitation que d'un cours sur l'assembleur et le langage machine.

Nous allons limiter cette section à la description de l'instruction *int*, qui va nous permettre de communiquer avec le système d'exploitation (pour les exemples, nous prendrons GNU/Linux).

L'instruction *int* a un comportement similaire à l'instruction *call*. Le registre *rip* va être modifié pour permettre l'exécution d'une procédure, mais l'adresse de l'instruction *int* est mémorisée, de telle sorte qu'à la sortie de la procédure, *rip* prend pour valeur l'adresse de l'instruction qui suit l'instruction *int*.

La syntaxe de l'instruction *int* est la suivante :

```
int Constante
```

Constante est un entier qui désigne le numéro du gestionnaire d'interruption à appeler. La valeur de certains registres sera utilisée par le gestionnaire d'interruptions afin de sélectionner la procédure appropriée et de lui transmettre ses paramètres.

Sous GNU/Linux, par exemple, *int $0x80* va appeler le gestionnaire d'interruption du système d'exploitation (le gestionnaire de numéro hexadécimal 80, soit 128 en décimal) . Le numéro de fonction système est donné par *%eax* et les paramètres sont transmis via les registres %ebx,%ecx, %edx,%esi,%edi (dans cet ordre). Lorsqu'il y a plus de 5 paramètres, *%ebx* contient tout simplement l'adresse des paramètres.

Voici par exemple, un petit programme en assembleur, destiné à être assemblé, lié et exécuté sous GNU/Linux :

```
       .data  # directive de création d'une zone de donnée
btlm:         # adresse symbolique pointant sur la chaîne:
      .string "Bonjour tout le monde!\n"
       .text  # directive de création d'une zone
              # d'instructions
       .globl main  # directive de création d'une étiquette
                    # de portée globale
main:         # main est l'adresse de début du programme
        movl    $4,%eax    # sélection de la fonction
                           # write du système
        movl    $1,%ebx    # dernier paramètre de write :
                           # stdout
        movl    $btlm,%ecx       # premier paramètre
                                 # de write : l'adresse de
                           # la chaîne de caractères à
                           # afficher
        movl    $23,%edx   # le nombre de caractères à
                           # afficher : 23
        int     $0x80      # appel de l'interruption
                           # 128 -> GNU/Linux
        movl    $1,%eax    # sélection de la fonction
                           # exit du système
        xorl    %ebx,%ebx  # mise à zéro du 1er paramètre
                           # en utilisant
                           # xor, c'est à dire ou exclusif
        int     $0x80      # appel de l'interruption
                           # 128 -> GNU/Linux
        ret          # fin du programme et retour au système
```

9. Autres instructions d'arithmétique entière

9.1 Multiplication et division sur des entiers non signés

9.1.1. Multiplication non signée

- *mul <x8>* effectue la multiplication du contenu du registre *al* avec le contenu de x8 qui est soit l'emplacement d'un octet, soit un registre de 8 bits. Le résultat est stocké dans le registre 16 bits *ax*.

- *mul <x16>* effectue la multiplication du contenu du registre *ax* avec le contenu de *x16* qui est soit l'emplacement d'un mot de 16 bits, soit un registre 16 bits. Le résultat est stocké dans *dx:ax*. Autrement dit, le mot (16 bits) de poids faible du résultat est stocké dans *ax* et le mot (16 bits) de poids fort dans *dx*. On peut se demander pourquoi le résultat n'est pas stocké dans *eax*. C'est pour une simple raison historique conservée par les contraintes de compatibilité : le registre *eax* n'existait pas dans le 8086.

- *mul <x32>* effectue la multiplication du contenu du registre *eax* avec le contenu de *x32* qui est soit l'emplacement d'un mot de 32 bits, soit un registre 32 bits. Le résultat est stocké dans *edx:eax*. Autrement dit, le mot (32 bits) de poids faible du résultat est stocké dans *eax* et le mot (32 bits) de poids fort dans *edx*.

- mul <x64> effectue la multiplication du contenu du registre *rax* avec le contenu de *x64* qui est soit l'emplacement d'un mot de 64 bits, soit un registre 64 bits. Le résultat est stocké dans *rdx:rax*. Autrement dit, le mot de 64 bits de poids faible du résultat est stocké dans *rax* et le mot de 64 bits de poids fort dans *rdx*.

- Les drapeaux OF et CF du registre d'état *RFLAGS* sont mis à 0 si la moitié supérieure des bits du résultat (*ah*, *dx*, *edx* ou *rdx*) est à 0, les deux drapeaux sont mis à 1 dans le cas contraire.

9.1.2. Division non signée

- *div <x8>* effectue la division de *ax* par *<x8>* (voir *mul*). Le quotient est stocké dans *al* et le reste dans *ah*.

- *div <x16>* effectue la division de *dx:ax* par *<x16>* (voir *mul*). Le quotient est stocké dans *ax* et le reste dans *dx*.

- *div <x32>* effectue la division de *edx:eax* par *<x32>* (voir *mul*). Le quotient est stocké dans *eax* et le reste dans *edx*.

- *div <x64>* effectue la division de *rdx:rax* par *<x64>* (voir *mul*). Le quotient est stocké dans *rax* et le reste dans *rdx.*

- Les drapeaux de *RFLAGS* ne sont pas affectés. En cas de dépassement, l'exception #DE (division par zéro) est générée[15].

*9.2 **Multiplication et division sur des entiers signés***

9.2.1 Multiplication signée

imul peut prendre :

- 1 opérande : Comportement similaire à *mul*.

- 2 opérandes : *imul <source>, <destination>*

Ici *<destination>* doit être un registre général dans lequel sera stocké le produit de *<source>* (valeur immédiate, registre ou emplacement mémoire) et de *<destination>*.

- 3 opérandes : *imul <source1>, <source2>, <destination>*

Ici *<destination>* doit être un registre général dans lequel sera stocké le produit de *<source1>* (registre ou emplacement mémoire) et de *<source2>* (valeur immédiate). CF et OF sont affectés de la même façon que pour *mul*.

9.2.2 Division signée

idiv fonctionne de la même façon que *div*.

15 Pour simplifier, disons qu'une exception est une erreur qui par défaut, sera gérée par le système d'exploitation.

10. Opérateurs logiques

1. *and* <*source*>, <*destination*> : Il s'agit du « ET bits à bits »
2. *or* <*source*>, <*destination*> : Il s'agit du « OU bits à bits »
3. *xor* <*source*>, <*destination*> : Il s'agit du « OU EXCLUSIF bits à bits »
4. not <source-destination> : Il s'agit du « NON bits à bits »

Comme pour les additions et les soustractions, l'opérande <destination> désigne aussi le résultat de l'opération. Par exemple :

```
movb   $0b11110000, %al    # 0b11110000 -> al
xorb   $0b01010101, %al    # 0b11110000 ou-exclusif
                           # 0b01010101 -> al
                           # ici al contient 0b10100101
```

ou encore :

```
       movb   $0b10101010, %al    # 0b10101010 -> al
       notb   %al                 # non(0b10101010) -> al
                           # ici al contient 0b01010101
```

11. Calculs en virgule flottante

11.1 Introduction

Nous avons vu que le microprocesseur n'est pas capable de travailler directement dans l'ensemble des nombres entiers naturels ou relatifs. Le processeur peut en revanche manipuler des sous-ensembles finis : entiers signés ou non signés représentés le plus souvent sur $8x2^n$ bits. Par conséquent, nos sous-ensembles sont définis très simplement par leurs bornes inférieures et supérieures.

Malheureusement, lorsque nous cherchons à représenter les nombres réels, les choses se compliquent : nous avons une infinité de valeurs entre une borne inférieure et une borne supérieure. Ceci pose évidemment un problème lorsqu'il s'agit de représenter ces nombres avec des valeurs binaires de taille fixe. Le choix de notre représentation va donc déterminer en plus des limites inférieures et supérieures, la précision maximale de nos calculs.

De même qu'un simple débordement lors d'une addition de nombres entiers peut produire un résultat final incorrect, **une très légère erreur de précision à un moment donné dans un calcul sur des nombres à virgule flottante peut mener à des résultats ultimes totalement erronés.**

11.2. La norme IEEE 754

Dans cette norme, les nombres sont représentés en plusieurs parties : le signe, l'*exposant* et la *mantisse*. C'est la représentation que nous adoptons naturellement en base décimale quand nous écrivons : -1234.10^{-9}. Or, -1234.10^{-9} peut s'écrire aussi -12340.10^{-10} ou $-123,4.10^{-8}$. Dans la première écriture, la mantisse a une taille de 4 chiffres et l'exposant s'écrit avec 1 chiffre, dans le 2e cas la mantisse a une taille de 5 chiffres et l'exposant s'écrit avec 2 chiffres, etc.

Pour simplifier les choses et gagner de la précision à taille mémoire constante, nos nombres seront normalisés à chaque fois que cela sera possible : notre mantisse aura toujours en base binaire la forme de *« 1 » « virgule » « fraction binaire »*. Notre exposant sera négatif pour les valeurs inférieures à 0 et positif ou nul pour les valeurs supérieures ou égales à 1.

Cette normalisation n'est bien entendu plus possible lorsque nous atteignons les bornes supérieures ou inférieures de l'exposant. Lorsque nous atteignons la limite inférieure de l'exposant, c'est-à-dire lorsque notre valeur (positive ou négative) est très proche de 0, nous dirons que le nombre est dénormalisé. L'exposant prendra la valeur 0 et la mantisse aura la forme « 0 » « virgule » « fraction binaire ». Lorsque nous dépassons la limite supérieure de l'exposant, notre nombre prendra une valeur qui symbolise +infini ou -infini.

Après une opération invalide dans les réels (racine d'un nombre négatif, division par 0), le résultat aura également une valeur remarquable. On peut voir ci-dessous la représentation 32 bits des nombres à virgule flottante. L'exposant (entier signé) n'est pas représenté en complément à 2 mais sous la forme *biaisée :* sur 8 bits, la valeur biaisée 127 correspond au zéro (non biaisé). Les valeurs biaisées qui vont de 1 à 126 correspondent aux valeurs non biaisées de -126 à -1 et les valeurs biaisées qui vont de 128 à 254 correspondent aux valeurs non biaisées de 1 à 126.

nombre	***Signe (1bit)***	***Exposant (8bits)***	***Mantisse (23 bits)***
« plus » zéro	0	0	0
« moins » zéro	1	0	0
dénormalisé positif	0	0	0b0,xxxx...xxxx
dénormalisé négatif	1	0	0b0,xxxx...xxxx
normalisé positif	0	1..0d254	0b1,xxxx...xxxx
normalisé négatif	1	1..0d254	0b1,xxxx...xxxx
+infini	0	0d255	0
-infini	1	0d255	0
±NaN (not a number)	0 ou 1	0d255	$x \neq 0$

11.3 Les registres du processeur à virgule flottante (x87)

Du point de vue du programmeur, le processeur à virgule flottante (FPU : floating point unit) est une unité distincte du processeur central (CPU: central processing unit). Il existe bien entendu des « ponts » entre les deux, mais chacun a ses propres registres et son propre jeu d'instructions. Ceci est encore un héritage du du 8086 qui était limité aux instructions sur les entiers signés et non signés, auquel on pouvait adjoindre un second processeur, le 8087 que l'on appelait coprocesseur arithmétique. Lorsque l'on a intégré les instructions et registres à virgule flottante, il a été décidé de préserver la compatibilité avec les logiciels écrits pour des machines à 2 processeurs 8086 et 8087.

Le FPU possède 8 registres de 80 bits chacun (1 bit pour le signe, 15 bits pour l'exposant, 64 bits pour la mantisse). Ces 8 registres ne sont pas adressables directement comme dans le cas de l'unité d'arithmétique entière. Ils constituent une pile de registres, que l'on nommera *%st(0), %st(1), ..., %st(7)*. *%st(0)* désigne le sommet de la pile (c'est-à-dire le dernier élément empilé) et *%st(i)* désigne le *i*-ème élément de la pile de registres.

11.4 Principales instructions de calcul

- fld, fst

L'instruction *fld* (float load) permet d'empiler un nombre à virgule flottante dans la pile des registres. La lettre qui suit indique la taille de l'opérande : *flds* pour empiler un nombre flottant de 32 bits, *fldl* pour empiler un nombre flottant de 64 bits et *fldt* pour empiler un flottant de 80 bits. Lorsqu'une valeur est empilée, elle est automatiquement convertie en nombre flottant de 80 bits. L'opérande qui suit la mnémonique *fld*, c'est-à-dire la valeur à empiler, peut être soit un registre *st(i)* soit un emplacement en mémoire. *fild* et *fist* (avec « i » pour *integer*) sont utilisés pour les conversions virgule flottante/entiers.

L'instruction *fst* (float store) copie la valeur contenue dans le sommet de la pile, *st(0)* à l'emplacement donné en opérande. *fstp* réalise la même opération en dépilant le sommet de la pile.

- fadd, fsub, fmul, fdiv

Ces instructions réalisent respectivement l'addition, la soustraction, la multiplication et la division de deux opérandes dont une au moins est un

emplacement de la pile de registres, l'autre opérande étant soit un registre de la pile, soit un emplacement en mémoire. Lorsqu'un seul opérande est spécifié, l'opération est réalisée entre l'opérande et le sommet de la pile *st(0)*. Le suffixe « p » peut être ajouté pour permettre un postdépilement des registres. Le suffixe « r » peut être ajouté aux mnémoniques fsub et fdiv afin d'inverser l'ordre des opérandes. Le résultat d'une opération est toujours empilé et se retrouve donc toujours dans %st(0).

Ici aussi, les suffixes « s », « l » ou « t » peuvent être ajoutés pour spécifier la taille des nombres flottants.

Par exemple : « *fdivrps diviseur* » va prendre 32 bits à l'adresse symbolique « diviseur », convertir ce nombre flottant sur 80 bits, diviser cette valeur par le sommet de la pile, remplacer le sommet de la pile par le résultat de la division et enfin dépiler la pile de registres.

- fiadd, fisub, fimul, fidiv

Ces instructions sont à utiliser lorsque l'emplacement mémoire contient un entier. Il sera converti en nombre flottant de 80 bits avant l'opération.

11.5 Comparaisons et branchements conditionnels

Le FPU possède son propre registre d'état et donc ses propres drapeaux. Lors d'une comparaison de 2 nombres à virgule flottante, les drapeaux C0, C1, C2 et C3 indiquent le résultat de la comparaison. Pour pouvoir utiliser les instructions classiques de branchement conditionnel, il faudra transférer les valeurs des drapeaux C0,...,C3 dans les drapeaux ZF, CF et PF du registre *RFLAGS*. Cependant pour les microprocesseurs récents, l'instruction *fcomi* effectue la comparaison et affecte directement les registres ZF, PF et CF, ce qui permet d'utiliser immédiatement après les instructions de branchement conditionnel classiques (jc, jnc, jo, jz, etc.). Après l'instruction « *fcomi %st(i)* » les trois registres de EFLAGS sont modifiés de la façon suivante :

Condition	*ZF*	*PF*	*CF*
st(0) > st(i)	0	0	0
st(0) < st(i)	0	0	1
st(0) = st(i)	1	0	0
non comparable	1	1	1

PF=1 par exemple lorsqu'un des deux registres a *NaN*[16] pour valeur. Si la comparaison s'est bien déroulée, alors PF =0 et les branchements conditionnels *je*, *ja*, *jb*, *jae*, *jbe*, *jna*, *jnb*, *jnae* et *jnbe* prennent la même signification que lorsqu'il suivent une comparaison de valeurs entières (voir chapitre 5).

12. Parallélisme (MMX, SSE, 3DNow!)

Les instructions de type MMX, SSE et 3DNow sont ce que l'on apelle des SIMD, pour *Single Instruction Multiple Data.* Une instruction va lancer une série d'opérations sur une série de données en parallèle. Nous allons voir les principaux concepts de programmation avec la technologie MMX. Les technologies SSE et 3DNow! étant de simples ajouts ou améliorations de MMX, nous n'allons pas les étudier ici.

12.1 ***Registres MMX***

Les registres MMX se superposent aux registres du FPU. Le programmeur ne pourra donc pas utiliser les instructions du FPU et celle de MMX simultanément. Il y a 8 registres MMX, nommés *mm0...*, *mm7*. Ils ont chacun une taille de 64 bits. Dans chacun de ces registres, nous pouvons stocker, au choix :

- 1 nombre entier de 64 bits

16 Not a Number (résultat d'une opération invalide)

- 2 nombres entiers de 32 bits
- 4 nombres entiers de 16 bits
- 8 nombres entiers sur 8 bits

Par exemple, si l'on veut stocker 8 entiers dans le registre *mm0*, ses bits numérotés de 0 à 7 contiendront le 1er nombre, ses bits numérotés de 8 à 15 contiendront le 2e nombre, etc.

12.2 ***Instructions MMX***

Avec le CPU ou le FPU, les différents problèmes qui pouvaient se produire lors d'un calcul (débordement, division par 0, etc.) étaient indiqués par des drapeaux ou par des exceptions. Dans le cas des instructions MMX, soit la retenue est ignorée (on obtient le modulo du résultat), soit les valeurs sont *saturées :* lorsqu'à la suite d'une opération, le résultat est supérieur à la borne maximale, il sera rendu égal à la borne maximale. Si le résultat est inférieur à la borne minimale, il sera rendu égal à la borne minimale. Cette manière de traiter les débordements est particulièrement bien adaptée aux algorithmes traitant de l'image, du son ou de la vidéo.

Les mnémoniques utilisées sont très proches de celles utilisées pour les opérations d'arithmétique entière classique. Pour les copies de données, on utilisera les instructions ***movd*** (32 bits) et ***movq*** (64 bits). Pour additionner de 8 octets à 8 autres octets en parallèle sans saturation s'écrira ***paddb <source>, <destination>.*** La destination doit être un registre MMX, la source est soit un registre MMX, soit un emplacement. Pour additionner en parallèle 4 nombres de 16 bits à 4 autres nombres de 16 bits, on écrira ***paddw.***

Enfin, une addition de 2x2 nombres de 32 bits en parallèle s'écrira ***paddd.*** L'addition avec saturation d'entiers signés s'écrira **padds** (avec « b » ou « w » en suffixe) et l' addition avec saturation d'entiers non signés s'écrira **paddus** (avec « b » ou « w » en suffixe)**.** Les mnémoniques correspondantes pour la soustraction sont ***psubb, psubw, psubd, psubsb, psubsw, psubusb*** et ***psubusw***.

pmull réalise la multiplication en parallèle des 4 entiers signés de 16 bits de la source aux 4 entiers signés de 16 bits de la destination et stocke la partie basse (16 bits) du résultat dans la destination. pmulh réalise le même calcul, mais stocke la partie haute du résultat. pmullw et pmulhw réalisent les mêmes

opérations avec 2x2 entiers signés de 32 bits.

Le jeu d'instructions MMX contient également des opérations de comparaison, de conversion, d'opération logiques et de décalage, mais nous avons vu les concepts principaux. Supposons qu'à l'adresse *photo*, nous ayons une image de 256x256 pixels en 256 niveaux de gris (0-> noir, 255 -> blanc). Voici un petit programme parallèle, 8 fois plus rapide que son équivalent séquentiel, qui augmente d'un cran la luminosité de l'image :

```
movq $photo,                    %rsi
movq $0x0101010101010101,       %mm0
                          # 8 octets de valeur 1 -> mm0
movw  $0,   %cx   # pixel courant

boucle:

movq  (%rsi), %mm1      # charge 8 octets -> mm1
paddusb %mm0, %mm1      # ajoute 1 à 8 pixels en //
movq   %mm1, (%rsi)     # transfère 8 pixels -> photo
addl  $8, %rsi          # avance de 8 pixels
addw  $8, %cx           # MAJ du compteur de pixels
jnc boucle              # boucle dans que cx <=65535
```

13. Bibliographie

- AMD64 Architecture Programmer's Manual. Volume 1 : application programming,

  http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf

- *Intel* ® Extended Memory 64 Technology — Porting IA-32 Applications to the Intel® Xeon® Processor

  http://www.intel.com/cd/ids/developer/asmo-na/eng/171850.htm?page=1

- Dean Elsner, Jay Fenlason & friends, Using as : The GNU Assembler, January 1994.

  http://www.gnu.org/software/binutils/manual/gas-2.9.1/

14. Exercices

14.1 Capacité des cases mémoires (voir section 1.1.)

14.1.1. QCM : Quel est le nombre maximal de valeurs distinctes que peut représenter une case mémoire de 3 bits ?

a. 3

b. 6 (2x3)

c. 7 (2^3-1)

d. 8 (2^3)

e. 9 (3^2)

Réponse : d. avec une case mémoire de 3 bits, on pourra représenter 2^3 valeurs distinctes. Représentées sous la forme de nombres binaires, il s'agit de 000, 001, 010, 011, 100, 101, 110 et 111.

14.1.2. QCM : Quelle est la taille minimale pour une case mémoire qui pourra représenter un mois particulier parmi les douze que compte un calendrier grégorien ?

a. 4 bits

b. 12 bits

b. 24 bits (2x12)

c. 144 bits (12^2)

d. 4096 bits (2^{12})

Réponse : a. Une case mémoire de 3 bits peut contenir une valeur parmi

seulement 8 (2^3) valeurs distinctes. Avec un bit supplémentaire (soit 4 bits), on peut par stocker une valeur parmi 16 valeurs distinctes. Pour stocker un mois particulier parmi 12 possibles, une taille de 4 bits est donc nécessaire et suffisante.

14.2. Poids des chiffres, parties hautes et basses d'un nombre (voir section 1.1.1)

14.2.1 quelle est la valeur en base décimale du nombre hexadécimal B105 ?

a. 11x16 + 1x32 + 0x48 + 5x64= 528

b. $11x16^3 + 1x16^2 + 0x16^1 + 5x16^0$=45333

c. 11x16x3 + 1x16x2 + 0x16x1 + 5x16x0= 560

d. 16 x ($11^3 + 1^2 + 0^1 + 5^0$) = 21344

Réponse : b.

14.2.2. Quel est le chiffre de poids fort et la partie basse de 3 chiffres du nombre AZ35T6 (exprimé dans une base numérique inconnue) ?

a. On ne peut pas répondre à cette question sans connaître la base numérique utilisée

b. chiffre de poids fort : 6 ; partie basse de 3 chiffres : AZ3

c. chiffre de poids fort : A ; partie basse de 3 chiffres : 5T6

d. chiffre de poids fort : A ; partie basse de 3 chiffres : AZ3

Réponse : a.

14.2.3. Quelle est la partie haute de 12 bits du nombre FEDCBA9 exprimé en base hexadécimale ?

a. FEDC

b. FED

c. FE

d. BA9

Réponse : b. (chaque chiffre hexadécimal nécessite 4 bits et seulement 4 bits pour son stockage).

14.2.4. Combien d'octets contient une case mémoire de 64 bits ?

a. 64

b. 6,4 (64/10)

c. 8 (64/8)

d. 512 (8*64)

Réponse : c. Un octet est un groupe – ou plutôt une séquence – de 8 bits.

14.3 À combien de cases mémoires distinctes peut-on accéder avec des adresses hexadécimales allant de 0 à FFFF ?

a. 2^{16}=65536 cases mémoires

b. 2^{8}=256 bits

c. 2^{4}=16 octets

d. 2^{16}-1=65535 kilo-octets

R**éponse :** a. chaque chiffre hexadécimal étant représenté sur 4 bits, l'adresse

est donc « codée » sur 16 bits. Il y a donc 2^{16} valeurs possibles pour une adresse.

14.4 Opérandes (voir section 1.3)

14.4.1 `movl nb1, al` est une instruction qui :

a) copie la valeur de 8 bits correspondant à l'étiquette nb1 dans le registre al

b) copie la valeur 32 bits contenue à l'adresse de l'étiquette nb1 dans le registre al

c) copie la valeur 32 bits contenue à l'adresse de l'étiquette nb1 dans la case mémoire dont l'adresse est représentée par l'étiquette al

d) copie la valeur du registre al dans le registre nb1

Réponse : c. Le registre 8 bits « al » est désigné par « %al ». La valeur de l'étiquette « nb1 » est désignée par « $nb1 ». S'il existait un registre nommé « nb1 », il faudrait s'y référer par « %nb1 ». En l'absence de signe indiquant un type d'opérande « constante » ($) ou « registre » (%), les étiquettes font référence à des cases mémoires définies symboliquement par le programmeur. Mov est la mnémonique qui correspond à une instruction de copie. Le « l » qui suit « mov » indique que les opérandes ont une capacité de 32 bits. L'opérande « source » est celle qui est à gauche de la virgule et l'opérande « destination », celle qui est à droite de la virgule (syntaxe AT&T).

14.5 Registres généraux (voir section 2.1)

14.5.1 parmi les propositions suivantes, lesquelles sont exactes ?

a) %bh et %bl sont des registres généraux d'une capacité de 8 bits ?

b) Toute modification du contenu de %cl entraine une modification du contenu de %ch.

c) Toute modification du contenu de %bl entraine une modification du contenu de %bx, %ebx et %rbx

d) Toute modification de %rdx entraine une modification du contenu de %edx, %dx, %dh et %dl

e) Une modification de %rax peut entrainer une modification de %eax, %ax, %ah ou %al

Réponse : a) Exact, par définition.

b) Faux. %cl et %ch font tous deux partie des registres %cx, %ecx et %rcx mais ils correspondent à des emplacements mémoires distincts.

c) Exact : %bl désigne les 8 bits de poids faible de %bx (partie basse de 16 bits de %rbx), mais aussi de %ebx (partie basse de 32 bits de %rbx) et donc de %rbx.

d) Faux: Par exemple, une modification qui n'affecterait que la partie haute de 32 bits de %rdx, laisserait inchangées les valeurs contenues dans %edx (partie basse de 32 bits de %rdx), dans %dx (partie basse de 16 bits de %rdx), dans %dh (partie haute de 8 bits de %dx) et dans %dl (partie basse de 8 bits de %rdx).

e) Exact : Si la modification de %rax affecte ses 16 bits de poids faible, alors elle affecte les registres %eax (32 bits de poids faible de %rax), %ax (16 bits de poids faibles), %ah (8 bits de poids fort de %ax) et %al (8 bits de poids faible de %rax).

14.6 Arithmétique entière (sections 2.2 et 2.3)

Donnez le résultat (en décimal) et les valeurs de CF (Carry Flag – entiers non signés) et OF (Overflow Flag – entiers signés) après les opérations ci-dessous. Pour vous éviter d'avoir recours à une calculatrice, voici les valeurs de quelques puissances de 2 : $2^7 = 128$; $2^8 = 256$; $2^{15}=32768$; $2^{16} = 65536$.

Taille opération	Opération	résultat signé	résultat non signé	CF = ?	OF = ?
8 bits	0-1	?	?	?	?
8 bits	127+1	?	?	?	?
8 bits	0xFF+1	?	?	?	?
8 bits	-128-1	?	?	?	?
16 bits	0-1	?	?	?	?
16 bits	32767+1	?	?	?	?
16 bits	0xFFFF+1	?	?	?	?
16 bits	-32768-1	?	?	?	?

Réponse:

Taille opération	Opération	résultat signé	résultat non signé	CF = ?	OF = ?
8 bits	0-1	-1	255	1	0
8 bits	127+1	-128	128	0	1
8 bits	0xFF+1	0	0	1	0
8 bits	-128-1	127	127	0	1
16 bits	0-1	-1	65535	1	0
16 bits	32767+1	-32768	32768	0	1
16 bits	0xFFFF+1	0	0	1	0
16 bits	-32768-1	32767	32767	0	1

Explication :

- Ligne 1 : 0b00000000 - 0b1 = 0b11111111, valeur qui est interprétée comme $2^{8\ -1}$ dans les entiers non signés (avec un Carry Flag à 1 qui indique une retenue, donc un résultat invalide) ou comme -1 dans les entiers signés (avec un Overflow Flag à 0 qui indique un résultat valide).

- Ligne 2 : 0b01111111 + 0b1 = 0b10000000, valeur qui est interprétée comme 2^7 dans les entiers non signés (avec un Carry Flag à 0 qui indique l'absence de retenue, donc un résultat valide) ou comme -2^7 dans les entiers signés (avec un Overflow Flag à 1 qui indique un résultat invalide).

- Ligne 3 : 0b11111111 + 0b1 = 0b00000000, valeur qui est interprétée comme 0 dans les entiers non signés (avec un Carry Flag à 1 qui indique une retenue, donc un résultat invalide) et également comme 0 dans les entiers

signés (mais cette fois avec un Overflow Flag à 0 qui indique un résultat valide).

- Ligne 4 : 0b10000000 - 0b1 = 0b01111111, valeur qui est interprétée comme 2^7-1 dans les entiers non signés (avec un Carry Flag à 0 qui indique l'absence de retenue, donc un résultat valide) et également comme 2^7-1 dans les entiers signés (mais cette fois avec un Overflow Flag à 1 qui indique un résultat invalide).

- Ligne 5 : 0b0000000000000000-0b1 = 0b1111111111111111, valeur qui est interprétée comme 2^{16} -1 dans les entiers non signés (avec un Carry Flag à 1 qui indique une retenue, donc un résultat invalide) ou comme -1 dans les entiers signés (avec un Overflow Flag à 0 qui indique un résultat valide).

- Ligne 6 : 0b0111111111111111+0b1 = 0b1000000000000000, valeur qui est interprétée comme 2^{15} dans les entiers non signés (avec un Carry Flag à 0 qui indique l'absence de retenue, donc un résultat valide) ou comme - 2^{15} dans les entiers signés (avec un Overflow Flag à 1 qui indique un résultat invalide).

- Ligne 7 : 0b1111111111111111+0b1 = 0b0000000000000000, valeur qui est interprétée comme 0 dans les entiers non signés (avec un Carry Flag à 1 qui indique une retenue, donc un résultat invalide) et également comme 0 dans les entiers signés (mais cette fois avec un Overflow Flag à 0 qui indique un résultat valide).

- Ligne 8 : 0b1000000000000000 – 0b1 = 0b0111111111111111, valeur qui est interprétée comme 2^{15}-1 dans les entiers non signés (avec un Carry Flag à 0 qui indique l'absence de retenue, donc un résultat valide) et également comme 2^{15}-1 dans les entiers signés (mais cette fois avec un Overflow Flag à 1 qui indique un résultat invalide).

14.7 La pile (section 7)

Remplacez les points d'interrogation par les valeurs adéquates :

movq $0x89, %rax	%rax = ?????	%rbx= <indéfini>	%rsp -> {}
movq $0x67, %rbx	%rax = ?????	%rbx = ?????	%rsp -> {}
pushq %rax	%rax = ?????	%rbx = ?????	%rsp-> {?????}
pushq %rbx	%rax = ?????	%rbx = ?????	%rsp-> {?????, ?????}
pushq $0x45	%rax = ?????	%rbx = ?????	%rsp -> {?????, ?????, ????}
popq %rax	%rax = ?????	%rbx = ?????	%rsp-> {?????, ?????}
popq %rax	%rax = ?????	%rbx = ?????	%rsp-> {?????}
popq %rbx	%rax = ?????	%rbx = ?????	%rsp -> {}

Réponse :

movq $0x89, %rax	%rax = $0x89	%rbx= <indéfini>	%rsp -> {}
movq $0x67, %rbx	%rax = $0x89	%rbx = $0x67	%rsp -> {}
pushq %rax	%rax = $0x89	%rbx = $0x67	%rsp-> {$0x89}
pushq %rbx	%rax = $0x89	%rbx = $0x67	%rsp-> {$0x67, $0x89}
pushq $0x45	%rax = $0x89	%rbx = $0x67	%rsp -> {$0x45, $0x67, $0x89}
popq %rax	%rax = $0x45	%rbx = $0x67	%rsp-> {$0x67, $0x89}
popq %rax	%rax = $0x67	%rbx = $0x67	%rsp-> {$0x89}
popq %rbx	%rax = $0x67	%rbx = $0x89	%rsp -> {}

15. Travaux dirigés

15.1. Instructions de copie

Prérequis : structure d'un programme, structure d'une instruction, type et taille des opérandes, direction des instructions, instruction mov

Donnez la valeur du registre %eax après l'exécution consécutive de chacune de ces instructions.

1. movl $0xEFFACE, %eax Réponse **:** 0xEFFACE

   explication : il s'agit d'une simple copie d'une constante (adressage immédiat) de 32 bits dans un registre de 32 bits.

2. movb $0b10, %ah Réponse **:** 0xEF02CE

   explication : %ah est la partie haute de 8 bits (2 chiffres hexadécimaux) du registre de 16 bits (4 chiffres hexadécimaux) %ax. %ax est lui-même la partie basse de 16 bits du registre %eax (32 bits, soit 8 chiffres hexadécimaux). La valeur binaire 0b10 contenue sur 1 octet (movb) correspond à un nombre hexadécimal de deux chiffres (0x02) qui va être copié et donc remplacer l'ancienne valeur (0xFA) contenue dans la partie %ah du registre %eax.

3. movw $10, %bx Réponse **:** 0xEF02CE

   explication : on copie la valeur 0x000A dans le registre %bx. %ax et %bx étant des registres séparés et indépendants, la valeur de %ax reste inchangée.

4. movw %bx, %ax Question: %ah=? Réponse **:** 0x00

   explication : Le registre %bx contient la valeur hexadécimale 000A. Cette valeur est copiée dans le registre %ax. Le registre %al étant la partie basse de 8 bits (2 chiffres hexadécimaux) de %ax, sa valeur devient 0A. Le registre %ah étant la partie haute de 8 bits de %ax, sa valeur devient 0x00.

15.2. Instructions additives

Prérequis : Ceux du 15.1 + 2.2 (entiers signés et non signés) + débordements

Donnez la valeur du registre %eax et des drapeaux OF, CF et ZF après l'exécution consécutive de chacune de ces instructions.

1. subl %eax, %eax Réponse **:** 0x00000000, ZF=1, CF=0, OF=0

   explication : %eax contient une valeur de 32 bits quelconque. L'instruction subl %eax, %eax réalise la soustraction de cette valeur et d'elle-même. Le résultat, zéro est copié dans l'opérande destination : %eax. Le drapeau ZF (Zero Flag) du registre d'état (RFLAGS) prend la valeur 1, indiquant que le résultat est égal à zéro.

2. subw $1, %ax Réponse **:** 0x0000FFFF, ZF=0, CF=1, OF=0

   explication : 0 — 1 = -1 . L'opération est réalisée sur 16 bits et sur le registre %ax, soit la partie basse de 8 chiffres hexadécimaux de %eax. 0xFFFF correspond à la valeur -1 codée en complément à deux. À l'issue de l'opération, le drapeau OF (Overflow) du registre d'état RFLAGS prend la valeur 0, indiquant la validité du résultat sur les entiers signés. Le drapeau CF (Carry Flag) prend quant à lui la valeur 1, indiquant un débordement sur les entiers non signés.

3. addl $2, %eax Réponse **:** 0x00010001, ZF=0, CF=0, OF=0

   explication : L'opération étant réalisé sur 32 bits et sur %eax, la retenue est propagée jusqu'au 5e chiffre le plus à droite de %eax. L'opération est valide sur les entiers signés et non signés.

4. addl $0FFEFFFF, %eax

Réponse **:** 0x10000000, ZF=0, CF=0, OF=0. **Explication :** La retenue est propagée jusqu'au chiffre hexadécimal le plus à gauche du registre 32 bits %eax. Le résultat est non nul, valide pour les entiers non signés et valide pour les entiers signés (ajout de 2 nombres positifs et résultat positif – bit de poids fort à 0). OF est donc positionné à 0.

15.3. Sauts conditionnels et inconditionnels (chapitre 3 du cours)

Quelle est la valeur de %ecx après l'exécution de ces petits programmes ? Il est dans certains cas impossible de déterminer cette valeur. Si c'est le cas, répondez par « indéfini ».

15.3.1

```
        movl    $0,     %ecx
Boucle: addl    $1,     %ecx
        jmp     Boucle
```

Réponse : indéfini.

Explication : À l'entrée de la boucle, %ecx contient la valeur 0. La valeur 1 est ajoutée à %ecx à chaque itération. Mais « jmp » étant un saut inconditionnel, nous avons à faire à une boucle infinie. Il est donc impossible de connaître la valeur de %ecx après l'instruction « jmp » puisque le pointeur d'instruction n'atteindra jamais cette adresse.

15.3.2

```
        movl    $0,     %ecx
Boucle: addl    $1,     %ecx
        jnc     Boucle
```

Réponse : 0.

Explication : À l'entrée de la boucle, %ecx contient la valeur 0. La valeur 1 est ajoutée à %ecx à chaque itération. « jnc Boucle » réalise un saut vers « addl $1, %ecx » lorsque le drapeau CF du registre d'état RFLAGS est à 0. Autrement dit, le saut sera effectué tant que l'opération « addl $1, %ecx » ne

provoque pas de débordement sur les entiers non signés. Lorsque %ecx passe de 0xFFFFFFFF à 0, CF passe à 1 et l'instruction « jnc » ne réalise pas le saut. L'exécution se poursuit donc à l'instruction qui suit immédiatement « jnc Boucle ». La valeur de %ecx est alors 0.

15.3.3

```
        movl    $0,     %ecx
Boucle: addl    $1,     %ecx
        jno     Boucle
```

Réponse : 0.

Explication : À l'entrée de la boucle, %ecx contient la valeur 0. La valeur 1 est ajoutée à %ecx à chaque itération. « jnc Boucle » réalise un saut vers « addl $1, %ecx » lorsque le drapeau OF du registre d'état RFLAGS est à 0. Autrement dit, le saut sera effectué tant que l'opération « addl $1, %ecx » ne provoque pas de débordement sur les entiers signés. Lorsque %ecx passe de 0x0FFFFFFF à 0x10000000, 0F passe à 1 et l'instruction « jno » ne réalise pas le saut. L'exécution se poursuit donc à l'instruction qui suit immédiatement « jnc Boucle ». La valeur de %ecx est alors 0x10000000.

15.3.4

```
(1)             movl    $0,     %ecx
(2)  Boucle:    movl    $210,   %eax
(3)             subl    %ecx    %eax
(4)             jc      Sortie
(5)             addl    $2,     %ecx
(6)             jmp     Boucle
(7)  Sortie:
```

Réponse : 212.

Explication : **À** l'entrée de la boucle (1), %ecx contient la valeur 0. La première instruction à l'intérieur de la boucle (2) consiste à copier la valeur décimale 0d210 dans le registre %eax. L'instruction suivante (3) soustrait la valeur courante de %ecx au registre %eax. Si la valeur courante de %ecx est strictement supérieure à 0d210, alors le drapeau CF est mis à 1 et l'instruction suivante (4) effectue un saut vers « Sortie » (7). Sinon, la valeur de %ecx est augmentée (5) de 2 et l'instruction « jmp » (6) renvoie l'exécution au début de boucle (2). %ecx ayant été initialisé à 0, il aura à chaque itération des valeurs paires. Par conséquent, l'opération de la ligne 3, provoquera la mise à 1 de CF (débordement d'un entier non signé) lorsque %ecx aura comme valeur 0d212.

15.4. Adressage indirect (section 4.2 du cours)

Ce petit programme permet de copier, octet par octet, une chaine de caractères terminée par 0, dont l'adresse est contenue dans le registre rsi vers une deuxième chaine de caractères dont l'adresse est contenue dans le registre %rdi. Remplacez chaque point d'interrogation par un caractère pertinent. Si nécessaire, vous pouvez aussi supprimer des points d'interrogation.

(1)	???????	mov?	?%rsi?,	?%al?
(2)		mov?	?%al?,	?%rdi?
(3)		j?	????????	
(4)	???	add?	??,	?%rsi?
(5)		add?	??,	?%rdi?
(6)		jmp	??????	
(7)	?????????			

Réponse :

(1)	Boucle:	movb	(%rsi),	%al
(2)		movb	%al,	(%rdi)
(3)		jz	FinCopie	
(4)		addq	$1,	%rsi
(5)		addq	$1,	%rdi
(6)		jmp	Boucle	
(7)	FinCopie:			

Correction :

- Ligne 1, Colonne 1 : Il faut une étiquette, car c'est le début de l'itération (copie d'un caractère)
- Ligne 1, Colonne 2 : Suffixe b, car la valeur copiée est un octet (8bits)
- Ligne 1, Colonne 3 : Il faut des parenthèses pour indiquer que la donnée à copier ne se trouve pas directement dans %rsi mais à l'adresse contenue dans %rsi
- Ligne 1, Colonne 4 : Il faut supprimer les ? : l'octet pointé par %rsi est copié directement dans %al
- Ligne 2, Colonne 2 : Suffixe b, car la valeur copiée est un octet (8bits)
- Ligne 2, Colonne 3 : Il faut supprimer les ? : la valeur à copier est contenue directement dans %al
- Ligne 2, Colonne 4 : Il faut des parenthèses pour indiquer que la valeur ne doit pas être copiée directement dans %rdi mais à l'adresse contenue dans %rdi
- Ligne 3, Colonne 2 : 'jz' : si l'octet qui vient d'être copié est égal à 0, la chaine de caractère d'origine est terminée, il faut donc sortir de la boucle.
- Ligne 3, Colonne 3 : Adresse symbolique définie à la ligne 7, c'est-à-dire à la sortie de la boucle.
- Ligne 4, Colonne 1 : Supprimer les ? : cette adresse indique le début de l'incrémentation des registres, mais n'étant pas utilisée comme cible d'un saut, il est parfaitement inutile de lui donner un nom.
- Ligne 4, Colonne 2 : Suffixe b, car la valeur ajoutée est sur un octet (8bits)
- Ligne 4, Colonne 3 : « $1 » : valeur constante désignant la taille (1 octet) de l'objet que l'on vient de copier, qu'il faut ajouter au registre %rsi pour qu'il pointe sur le prochain octet à copier

- Ligne 4, Colonne 4 : Supprimer les ? : C'est bien l'adresse qu'il faut augmenter et cette adresse est contenue directement dans %rsi

- Ligne 5, Colonne 2 : Suffixe b, car la valeur ajoutée est sur un octet (8bits)

- Ligne 5, Colonne 3 : « $1 » : valeur constante désignant la taille (1 octet) de l'emplacement où l'on a copié, qu'il faut ajouter au registre %rdi pour qu'il pointe sur le prochain emplacement de 8 bits

- Ligne 5, Colonne 4 : Supprimer les ? : C'est bien l'adresse qu'il faut augmenter et cette adresse est contenue directement dans %rdi

- Ligne 6, Colonne 3 : Etiquette définie en ligne 1

Explication :

(1) Copie de l'octet [movb] pointé [()] par le registre 64 bits %rsi dans le registre 8 bits %al.

(2) Copie de l'octet contenu dans %al à l'emplacement pointé [()] par le registre 64 bits %rdi. **Note :** %al joue le rôle de registre intermédiaire, car la copie directe d'une valeur pointée vers une valeur pointée n'est pas implémentée dans les processeurs de la famille 80x86

(3) Si l'octet courant, contenu dans %al est égal à zéro (ZF=1), la copie est terminée (poursuite de l'exécution du programme à la ligne 7). Sinon (ZF=0), l'exécution se poursuit à la ligne 4.

(4) L'adresse d'origine (contenue dans %rsi) de la copie est augmentée de 1. Autrement dit, %rsi pointe sur le prochain octet à copier.

(5) L'adresse de destination (contenue dans %rdi) de la copie est augmentée de 1. Autrement dit, %rdi pointe sur le prochain octet à modifier.

(6) Retour à la ligne 1, c'est à dire copie de l'octet pointé par la nouvelle valeur de %rsi à la nouvelle adresse contenue dans %rdi.

(7) Fin de la copie. Cette ligne ne peut être atteinte que lorsque ZF=1 à la

ligne 3.

15.5. Adressage indirect indexé (4.3)

Dans cet exercice, nous voulons calculer le poids total d'une suite d'objets dont les caractéristiques sont stockées en mémoire sous la forme d'une suite de cases mémoires contiguës : n, t, p, v... t, p, v où n est un entier non signé 8 bits représentant le nombre d'objets, t est un entier non signé 32 bits représentant la taille de l'objet, p est un entier non signé 16 bits représentant le poids de l'objet, et v est un entier non signé de 64 bits représentant le volume de l'objet. Cette suite de cases mémoires est pointée par %rsi et le poids total devra être stocké dans %rax. Remplacez les points d'interrogation pour obtenir un programme correct.

(1)		movq	$0,	??%rax?	# Poids total à zéro
(2)		movq	$0,	??%rbx?	# Poid courant à zéro
(3)		movb	??%rsi?,	??%cl?	# copie le nombre total d'objets dans le registre %cl
(4)		addq	$1,	??%rsi?	# %rsi pointe sur l'objet courant
(5)	Boucle:	movw	??%rsi?,	??%bx?	# copie le poids de l'objet

					courant dans %rbx
	(6)		addq	??%rbx?	??%rax? # ajoute le poids de l'objet courant à %rax
	(7)		addq	???,	??%rsi? # fait pointer %rsi sur l'objet suivant
	(8)		subb	$1,	??%cl? # %cl contient le nombre d'objets restant
	(9)		jnz	Boucle	# nouvelle itération s'il reste des objets à traiter
	(10)	Fin:			

Réponse :

```
(1)             movq    $0,     %rax    # Poids total à zéro
(2)             movq    $0,     %rbx    # Poid courant à zéro
(3)             movb    (%rsi), %cl     # copie le nombre total d'objets dans le registre %cl
(4)             addq    $1,     %rsi    # %rsi pointe sur l'objet courant
(5)     Boucle: movw    4(%rsi),%bx     # copie le poids de l'objet courant dans %rbx
(6)             addq    %rbx    %rax    # ajoute le poids de l'objet courant à %rax
(7)             addq    $14,    %rsi    # fait pointer %rsi sur l'objet suivant
```

(8)		subb	$1,	%cl	# %cl contient le nombre d'objets restant
(9)		jnz	Boucle		# nouvelle itération s'il reste des objets à traiter
(10)	Fin:				

Correction :

➔ Ligne 1 : Il s'agit d'un accès direct à %rax, qui par ailleurs n'est pas initialisé. Utiliser ici les parenthèses reviendrait à mettre à 0 une case mémoire aléatoirement pointée par %rax.

➔ Ligne 2 : Il s'agit d'un accès direct à %rbx, qui par ailleurs n'est pas initialisé. Utiliser ici les parenthèses reviendrait à mettre à 0 une case mémoire aléatoirement pointée par %rbx.

➔ Ligne 3 : %rsi doit être entouré de parenthèses (sans index) puisqu'il contient l'adresse précise dans laquelle est stockée le nombre d'objets. %cl ne doit quant à lui pas être entouré de parenthèses puisqu'il doit stoker directement cette valeur.

➔ Ligne 4 : Pas de parenthèses puisque %rsi doit contenir directement l'adresse de l'objet courant.

➔ Ligne 5 : %rsi pointe sur l'objet courant, c'est-à-dire sur un entier de 32 bits (4 octets) indiquant la taille de l'objet. 4(%rsi) est donc le contenu de la case mémoire pointée par %rsi+4, c'est-à-dire le poids de l'objet courant. Il ne faut pas de parenthèses autour de %bx puisque ce registre doit contenir directement le poids de l'objet courant.

➔ Ligne 6 : Aucune parenthèse puisque %rbx contient directement le

poids de l'objet courant et que %rax doit contenir directement le poids global des objets jusqu'à l'objet courant.

➔ Ligne 7 : Pour faire pointer %rsi sur l'objet suivant, il faut ajouter à l'adresse qu'il contient, la taille en octets de l'élément courant, soit 4 (taille) + 2 (poids) + 8 (volume) = 14 octets. %rsi doit contenir directement cette adresse et il ne doit donc pas être entouré de parenthèses.

➔ Ligne 8 : %cl contient directement le nombre d'objets restants à traiter et il ne doit donc pas être entouré de parenthèses.

Explication : voir commentaires dans le programme

15.6. Indexations complexes (section 4.4 du cours)

Dans cet exercice, la question posée est la même que dans l'exercice 15.5, mais le programme est plus court, car il utilise les possibilités d'indexation complexe du processeur. Nous voulons calculer le poids total d'une suite d'objets dont les caractéristiques sont stockées en mémoire sous la forme d'une suite de cases mémoires contiguës : n, t, p, v, ... t, p, v où n est un entier non signé 8 bits représentant le nombre d'objets, t est un entier non signé 32 bits représentant la taille de l'objet, p est un entier non signé 16 bits représentant le poids de l'objet, et v est un entier non signé de 16 bits représentant le volume de l'objet. Cette suite de cases mémoires est pointée par %rsi et le poids total devra être stocké dans %rax. Remplacez les points d'interrogation pour obtenir un programme correct.

```
(1)     movq    $0,     %rax    # Poids total à zéro
(2)     movq    $0,     %rbx    # Poid courant à zéro
(3)     movq    $0,     %rcx    # mets à zéro la partie haute de %rcx
```

(4)		movb	(%rsi),	%rcx	# copie le nombre total d'objets dans le registre %rcx
(5)	Boucle:	????	??,	%rcx	# %rcx contient le numéro courant de l'objet à traiter (0 si un seul objet)
(6)		movw	?(%rsi, %rcx, ?),	%bx	# copie le poids de l'objet courant dans %rbx
(7)		addq	%rbx	%rax	# ajoute le poids de l'objet courant à %rax
(8)		jnz	Boucle		# nouvelle itération s'il reste des objets à traiter
(9)	Fin:				

Réponse :

```
(1)             movq    $0,             %rax    # Poids total à zéro
(2)             movq    $0,             %rbx    # Poid courant à zéro
(3)             movq    $0,             %rcx    # mets à zéro la partie haute de %rcx
(4)             movb    (%rsi),         %rcx    # copie le nombre total d'objets dans le registre %rcx
(5)     Boucle: subq    $1,             %rcx    # %rcx contient le numéro courant de l'objet à traiter (0 si un seul objet)
(6)             movw    5(%rsi, %rcx, 8),%bx    # copie le poids de l'objet courant dans %rbx
(7)             addq    %rbx            %rax    # ajoute le poids de
```

				l'objet courant à %rax
(8)		jnz	Boucle	# nouvelle itération s'il reste des objets à traiter
(9)	Fin:			

Explication :

%rsi pointe sur le nombre d'objets (1 octet), %rsi+1 pointe donc sur la taille du premier objet. Si l'on considère que le premier objet correspond à %rcx=0, le deuxième à %rcx=1, etc et puisque l'on sait que la taille de chaque objet est de 8 octets (taille sur 4 octets, poids sur 2 octets et volume sur 2 octets), l'adresse de l'objet courant est donnée par %rsi+1+8*%rcx. Pour avoir l'adresse du poids de l'objet courant, il suffit de rajouter les 4 octets pris par la taille de l'objet, soit : %rsi+1+4+8*%rcx. Pour accéder au contenu de cette adresse, il suffit donc d'écrire 5(%rsi, %rcx, 8).

En ligne 4, il faut s'assurer que le premier objet à traiter corresponde bien à une valeur de 0 pour %rcx. C'est pourquoi la décrémentation de %rcx est réalisée en début d'itération.

15.7. Algorithme de tri « Bulle » (chapitres 5 et 6 — comparaisons et structures).

On considère un tableau d'entiers signés sur 16 bits, pointé par « a » et l'algorithme de tri suivant :

```
PROGRAMME Tri_Bulle
      VARIABLE permut : Booleen;
        REPETER
           permut = FAUX
           POUR i VARIANT DE 0 à N-1 FAIRE
             SI a[i] > a[i+1] ALORS
                   echanger a[i] et a[i+1]
                   permut = VRAI
             FIN SI
            FIN POUR
         TANT QUE permut = VRAI
      FIN PROGRAMME
```

Il s'agit maintenant de traduire ce programme en assembleur (64 bits).

1. Combien d'«étiquettes symboliques» nous faudra traduire en assembleur ? [QCM : 0, 1, 2, 3 , 4, 5]

Réponse : 4 : a, i, N, permut

2. Quel registre pour a ? [QCM : %rax, %bx, %cl, %rsi, %edi ?]

Réponse : N'importe quel registre 64 bits peut stocker l'adresse du tableau, mais par convention, on choisira **%rsi** (source index).

3. Quel registre pour i ? [QCM : %rax, %bx, %rcx, %dl, %esi, %di]

Réponse : N'importe quel registre général pourrait faire l'affaire, mais si l'on veut pouvoir traiter des tableaux de grande taille, on choisira un registre 64 bits. Aussi, par convention on choisira **%rcx** (compteur).

4. Quel registre pour N ? [QCM : %eax, %bx, %rcx, %dl, %rsi, %di, valeur immédiate]

Réponse : N est une constante. Nous n'utiliserons pas un registre, mais simplement une **valeur immédiate**.

5, Quel registre pour permut ? [QCM : %al, %bl, %cx, %dl, %esi]

Réponse : Nous n'avons besoin que d'un seul bit, mais quitte à « gaspiller » 7 bits, il sera plus simple d'utiliser un registre général 8 bits . %rcx est déjà utilisé et %rax nous servira très probablement de stockage temporaire. Nous pouvons donc choisir **%bl ou %dl**.

6. Les lignes du programme assembleur qui correspond à l'algorithme de tri on été mélangées. Donnez l'ordre correct des lignes :

(1)	Boucle_Pour:	cmpq	$49,	%rcx	# **à N-1**
(2)		cmpw	%ax,	2(%rsi, %rcx, 2)	# compare a[i] et a[i+1]
(3)		movw	2(%rsi, %rcx, 2),	%bx	# bx contient a[i+1]
(4)		jge	Fin_Si		**# SI a[i] > a[i+1] ALORS**
(5)		jg	Fin_Pour		
(6)					# echanger a[i] et a[i+1] :
(7)		movw	%ax,	2(%rsi, %rcx, 2)	# ancien a[i] -> a[i+1]
(8)		jnz	REPETER		**# TANT QUE permut =**

					VRAI
(9)	REPETER:	movb	$0,	%bl	# permut = FAUX
(10)		movw	%bx,	(%rsi, %rcx, 2)	# a[i+1] -> a[i]
(11)		movb	$1,	%bl	# permut = VRAI
(12)	Fin_Si:	addq	$1,	%rcx	# i=i+1
(13)		movq	(%rsi, %rcx, 2),	%ax	# ax contient a[i]
(14)		movq	$0,	%rcx	# **POUR i VARIANT DE 0**
(15)		jmp	Boucle_Pour		
(16)	Fin_Pour:	cmpb	$0,	%bl	# compare bl à FAUX

Réponse :

(9)	REPETER:	movb	$0,	%bl	# permut = FAUX
(14)		movq	$0,	%rcx	# **POUR i VARIANT DE 0**
(1)	Boucle_Pour:	cmpq	$49,	%rcx	# **à N-1**
(5)		jg	Fin_Pour		
(13)		movq	(%rsi, %rcx, 2),	%ax	# ax contient a[i]
(2)		cmpw	%ax,	2(%rsi, %rcx, 2)	# compare a[i] et a[i+1]
(4)		jge	Fin_Si		**# SI a[i] > a[i+1] ALORS**
(3)		movw	2(%rsi, %rcx, 2),	%bx	# bx contient a[i+1]
(6)					# echanger a[i] et a[i+1] :
(10)		movw	%bx,	(%rsi, %rcx, 2)	# a[i+1] -> a[i]
(7)		movw	%ax,	2(%rsi, %rcx, 2)	# ancien a[i] -> a[i+1]
(11)		movb	$1,	%bl	# permut = VRAI
(12)	Fin_Si:	addq	$1,	%rcx	# i=i+1

(15)		jmp	Boucle_Pour		
(16)	Fin_Pour:	cmpb	$0,	%bl	# compare bl à FAUX
(8)		jnz	REPETER		**# TANT QUE permut = VRAI**

15.8. Tri « bulle » procédural (sections 7 et 8).

En reprenant l'algorithme du tri « bulle », nous écrivons une procédure qui réalise ce tri. Les paramètres de la procédure sont passés par la pile. La procédure « TriBulle » fait elle-même appel à une sous-procédure nommée « FaitPermutations » qui fait elle-même appel à une procédure « Permute ». Le programme principal appelle la procédure « TriBulle » pour classer par poids croissant les objets décrits dans l'exercice 15.6. Insérer dans le programme ci-dessous, les instructions : call TriBulle, call FaitPermutations, call Permute.

(1)	main:	pushq	$Stock		# empile l'adresse du tableau d'objets
(2)	fin:	retq			# retour au système d'exploitation
(3)	TriBulle:	movq	8(%rsp),	%rsi	# %rsi pointe sur le tableau à trier
(4)		movq	$0,	%rdx	# pour mettre à zéro la partie haute de %rdx
(5)		movb	(%rsi),	%dl	# %rdx contient le nombre d'objets à trier (N)
(6)		subq	$1,	%rdx	# %rdx est la limite de la boucle Pour (N-1)
(7)	Repeter:				
(8)		cmpb	$0,	%bl	# %bl = 0 si pas de nouvelle permutation
(9)		jne	Repeter		# tant que permut = VRAI
(10)		retq	$8		# retour à l'appel avec dépilement du paramètre d'appel

(11)	FaitPermutations:	movb	$0,	%bl	# permut = FAUX
(12)	PourInit:	movq	$0,	%rcx	# Pour i=0
(13)	PourTest:	cmpq	%rdx,	%rcx	# Jusqu'à n-1
(14)		jae	FinPour		# on sort de la boucle quand i>=n-1
(15)	Si:	movw	5(%rsi, %rcx, 8),	%ax	# compare T[i].poids
(16)		cmpw	%ax,	13(%rsi, %rcx, 8)	# avec T[i+1].poids
(17)		jae	FinSi		
(18)		movb	$1,	%bl	# permut = VRAI
(19)	FinSi:	add	$1,	%rcx	# élément suivant
(20)		jmp	PourTest		
(21)	FinPour:	retq			# retour à l'appel
(22)	Permute:	pushq	%rdx		# sauve le nombre d'objets à trier
(23)		movq	1(%rsi, %rcx, 8),	%rax	# sauve T[i]
(24)		movq	9(%rsi, %rcx, 8),	%rdx	# sauve T[i+1]
(25)		movq	%rdx,	1(%rsi, %rcx, 8)	# copie T[i+1] dans T[i]

(26)		movq	%rax,	9(%rsi, %rcx, 8)	# copie ancien T[i] dans T[i+1]
(27)		popq	%rdx		# restaure le nombre d'objets à trier
(28)		retq			

Question :

- call TriBulle : entre la ligne ? et la ligne ?
- call FaitPermutations : entre la ligne ? et la ligne ?
- call Permute : entre la ligne ?? et la ligne ??

Réponse :

- call TriBulle : entre la ligne 1 et la ligne 2
- call FaitPermutations : entre la ligne 7 et la ligne 8
- call Permute : entre la ligne 17 et la ligne 18

15.9. Plus Petit Commun Multiple (Chapitre 8.2, 9 et 10)

Le plus petit commun multiple (P.P.C.M) de deux entiers non nuls a et b peut être calculé comme la valeur absolue du produit de a et b divisé par le plus grand commun diviseur (ou P.G.C.D.) de a et b.

En utilisant l'algorithme d'Euclide pour le calcul du PGCD, nous avons l'algorithme suivant :

```
Fonction PGCD(a:nombre, b:nombre):nombre
Si b=0
| alors PGCD=a
Sinon
| r egal au reste de la division (entière) de a par b
| PGCD=PGCD(b, r)
Finsi
```

et

```
Fonction PPCM(a:nombre, b:nombre):nombre
PPCM=a*b
Si PPCM < 0
| alors PPCM=-PPCM
FinSi
PPCM=PPCM*PGCD(a,b)
```

Remettez dans l'ordre les lignes des programmes assembleur correspondant à ces 2 fonctions :

PGCD :

1	AlorsPGCD:			
		movq	%rax, 16(%rsp)	# remplace le 1er paramètre par le résultat du PGCD
		retq	$8	# retourne en dépilant le 2e paramètre

2	PGCD:			
		movq	16(%rsp), %rax	# Le nombre a
		movq	8(%rsp), %rbx	# Le nombre b

3	SinonP GCD:			
		movq	$0, %rdx	# initialise partie haute de 64 bits du numérateur
		divq	%rbx	# divise rdx:rax par rbx, %rax contient

4		cmpq	$0, %rbx	
		je	AlorsPGCD	# si b=0 on sort en renvoyant a

5		popq	16(%rsp)	# résultat du PGCD à la place du 1er paramètre
		retq	$8	# retourne en dépilant le 2e paramètre

6		pushq	%rbx	# empile b
		pushq	%rdx	# empile le reste r
		call	PGCD	# appel PGCD(b,r)

PPCM :

1		movq	$0, %rdx	# initialise partie haute de 64 bits du numérateur
		divq	%rbx	# %rax : quotient de a*b/PGCD(a,b)
		movq	%rbx, 16(%rsp)	# remplace le 1er paramètre par le résultat du PGCD

2		movq	16(%rsp)	# Le nombre a
		movq	8(%rsp)	# Le nombre b

	PPCM:	pushq	16(%rsp)	# Le nombre a
3		pushq	8(%rsp)	# Le nombre b
		call	PGCD	

4		retq	$8	# sort en dépilant le 2e paramètre

5		mulq	%rbx	# rax contient a*b
		popq	%rbx	# rbx contient PGCD(a,b)

Réponse :

2	PGCD:			
		movq	16(%rsp), %rax	# Le nombre a
		movq	8(%rsp), %rbx	# Le nombre b
4		cmpq	$0, %rbx	
		je	AlorsPGCD	# si b=0 on sort en renvoyant a
3	SinonPGCD:			
		movq	$0, %rdx	# initialise partie haute de 64 bits du numérateur
		divq	%rbx	# divise rdx:rax par rbx, %rax contient
6		pushq	%rbx	# empile b
		pushq	%rdx	# empile le reste r
		call	PGCD	# appel PGCD(b,r)

5		popq	16(%rsp)	# résultat du PGCD à la place du 1er paramètre
		retq	$8	# retourne en dépilant le 2e paramètre
1	AlorsPGCD:			
		movq	%rax, 16(%rsp)	# remplace le 1er paramètre par le résultat du PGCD
		retq	$8	# retourne en dépilant le 2e paramètre

3	PPCM:	pushq	16(%rsp)	# Le nombre a
		pushq	8(%rsp)	# Le nombre b
		call	PGCD	

2		movq	16(%rsp)	# Le nombre a
		movq	8(%rsp)	# Le nombre b
5		mulq	%rbx	# rax contient a*b
		popq	%rbx	# rbx contient PGCD(a,b)
1		movq	$0, %rdx	# initialise partie haute de 64 bits du numérateur
		divq	%rbx	# %rax : quotient de a*b/PGCD(a,b)
		movq	%rbx, 16(%rsp)	# remplace le 1er paramètre par le résultat du PGCD
4		retq	$8	# sort en dépilant le 2e paramètre

15.10. Calcul arithmétique flottant (chapitre 11)

Vous trouverez ci-après la formule mathématique de calcul d'une mensualité de crédit immobilier :

$$\frac{K \times \frac{t}{12}}{1-(1+\frac{t}{12})^{-n}}$$

où :

— m est la mensualité
— K est le capital emprunté
— t est le taux annuel proportionnel
— n est le nombre de mensualités

Le programme assembleur suivant est une procédure qui permet de réaliser ce calcul en considérant que m, K, t et n sont des flottants codés sur 64 bits. Vous devez simplement remplacer les lettres en rouge par des valeurs entières comprises entre 0 et 7.

```
main:
################# Calcul de m ######
        fldl    t                # st(0) = t
        fildl   NBMOISDANSANNEE  # st(0)=12 ; st(1)=t
        fdivrp  %st(a),%st(b)    # st(0)=t/12
        fldl    k                # st(0)=K ; st(1) = t/12
        fmul    %st(c), %st(d)   # st(0)=K.t/12 ; st(1) = t/12
        fld1                     # st(0)=1.0 ; st(1)=K.t/12 ; st(2) = t/12
        fadd    %st(e), %st(f)   # st(0)=1.0+t/12 ; st(1)=K.t/12 ;
                                 #st(2) = t/12
        movq    n,       %rcx    # le nombre de mensualités
        fld1                     # st(0)=1.0 ; st(1)=1.0+t/12 ;
st(2)=K.t/12 ; st(3) = t/12
PuissanceN:
        fdiv    %st(g), %st(h)   # st(0)=(1.0+t/12)^-rcx ;
st(1)=1.0+t/12 ; st(2)=K.t/12 ; st(3) = t/12
        subq    $1,      %rcx    # rcx=rcx-1
        jnz     PuissanceN       # vers puissance suivante
        fld1                     # st(0)=1.0 ; st(1)=(1.0+t/12)^n ;
st(2)=1.0+t/12 ; st(3)=K.t/12 ; st(4) = t/12
        fsubp   %st(i), %st(j)   # st(0)=1.0-(1.0+t/12)^n ;
st(1)=1.0+t/12 ; st(2)=K.t/12 ; st(3) = t/12
        fdivrp  %st(k), %st(l)   # st(0)=1.0+t/12 ; st(1)=
                                 # (K.t/12)/(1.0-(1.0+t/12)^n) ;
                                 # st(2) = t/12
```

```
        fstp    %st(m)          # st(0)=(K.t/12)/(1.0-(1.0+t/12)^n) ;
                                # st(1) = t/12
        fstpl   m               # stocke le résultat ; st(0)=t/12
        fstp    %st(n)          # vide la pile des registres flottants

################ Affiche le résultat #######

        pushq m                 # sauve m
        movsd   m,      %xmm0
        movl    $AfficheM, %edi  # chaine de format
        movl    $1, %eax        # stdout
        call    printf
        addq    $8, %rsp        # libère m

        movq    $1,%rax         # sélection de la fonction exit du
système
        xorq    %rbx,%rbx       # mise à zéro du 1er paramètre en
utilisant
                                # xor, c'est à dire ou exclusif
        int     $0x80           # appel de l'interruption
                                # 128 -> GNU/Linux
fin:    ret
```

Réponse : **a**=0; **b**=1; **c**=1; **d**=0; **e**=2; **f**=0; **g**=1; **h**=0; **i**=0; j=1; **k**=0; **l**=2; **m**=0; **n**=0 (voir aussi correction complète en section (16.2.4)

15.11. Produit scalaire via instructions SIMD.

Il s'agit ici d'écrire une procédure réalisant le produit scalaire de 2 matrices 4x4 d'entiers signés sur 16 bits. Le résultat de l'opération est une matrice 4x4 d'entiers signés sur 32 bits. Afin d'obtenir les meilleures performances en temps de calcul, nous utilisons les instructions SIMD (single instructions multiple data ; instruction unique à données multiples).

Voici la principale procédure de ce programme :

```
ProduitScalaire:
        movq    16(%rsp), %rdi  # adresse de la deuxième matrice
        pushq   %rdi            # pivote la 2e matrice pour avoir les
vecteurs
                                # colonnes en lignes
        call    Pivote
        movq     8(%rsp), %rcx  # adresse de la matrice produit
        movq    16(%rsp), %rdi  # adresse de la deuxième matrice
        movq    24(%rsp), %rsi  # adresse de la première matrice
        movq    $0, %rax        # indice de la ligne de la première
matrice
TraiteLigne:
        movq    $0, %rbx        # indice de la ligne de la transposée
                                # de la seconde matrice
```

```
TraiteColonne:
        movq    (%rsi, %rax, 8), A      # charge la ligne de la 1ere
                                        # matrice dans  A
        movq    (%rdi, %rbx, 8), B      # charge la colonne de la 2e
                                        # matrice (ou ligne de
                                        # sa transposée) dans  B
        C       B, A                    # A : partie basse du produit
                                        # composante par composante
        call    SommeDesComposantes     # rdx contient la somme des
                                        # composantes 8 bits de A

        movw    %dx, (%rcx)             # copie du résultat dans la
                                        # matrice produit
        movq    (%rsi, %rax, 8), A      # charge la ligne de la 1ere
                                        # matrice dans  A
        D       B, A                    # A : partie haute du produit
                                        # composante par
                                        # composante
        call    SommeDesComposantes     # rdx contient la somme des
                                        # composantes 8 bits de %mm0
        shll    $16, %edx               # décale le résultat de 16 bits
                                        # vers la gauche pour
                                        # le mettre en partie haute

        addl    %edx, (%rcx)            # ajoute la partie haute du
                                        # résultat dans la
                                        # matrice produit
        addq    $4, %rcx                # pointe sur l'élément suivant à
                                        # calculer
        addq    $1, %rbx                # indice de colonne suivant
        cmpq    $4, %rbx                # fin de colonne ?
        jne     TraiteColonne
        movq    $0, %rbx                # on recommence à la colonne 0
        addq    $1, %rax                # ligne suivante
        cmpq    $4, %rax                # dernière ligne ?
        jne     TraiteLigne
        ret $24
```

Exercice : Remplacez **A**, **B**, **C** et **D** par les registres ou les mnémoniques SIMD qui permettent d'obtenir un programme correct.

Réponse :

- **À** = %mm0
- **B** = %mm1
- **C** = pmullw
- **D** = pmulhw

16. Travaux pratiques : Programmer en Assembleur sous GNU/Linux

Les programmes ci-dessous pourront être exécutés avec une distribution de GNU/Linux 64 bits, avec les logiciels g++ et insight. Ils pourront être facilement adaptés pour fonctionner sou GNU/Linux 32 bits.

16.1 Premiers pas

1. Ouvrez un terminal. Vous pouvez y accéder en cliquant sur le menu principal de votre distribution. Il se trouve dans la catégorie *applications* et dans la sous-catégorie *système*.

2. Vous taperez toutes vos commandes dans la fenêtre du terminal. Commencez par créer un répertoire de travail « mkdir tp_asm » et choisissez-le comme répertoire courant « cd tp_asm ».

3. À ce stade, vous avez probablement un choix à faire parmi une grande quantité d'éditeurs disponibles. Je choisis emacs pour mes exemples. Pour éditer votre premier programme assembleur, lancez l'éditeur : « emacs btlm.s ». Faites un copier-coller du code donné en section 8.2 :

```
.data                   # directive de création d'une zone de donnée
btlm:                   # adresse symbolique pointant sur la chaîne:
      .string "Bonjour tout le monde!\n"
        .text           # directive de création d'une zone
                        # d'instructions
        .globl main     # directive de création d'une étiquette
                        # de portée globale
main:                   # main est l'adresse de début du programme
        movl    $4,%eax   # sélection de la fonction write du système
        movl    $1,%ebx   # dernier paramètre de write : stdout
        movl    $btlm,%ecx # premier paramètre de write : l'adresse de
                        # la chaîne de caractères à afficher
        movl    $23,%edx  # le nombre de caractères à afficher : 23
        int     $0x80     # appel de l'interruption 128 -> GNU/Linux
        movl    $1,%eax   # sélection de la fonction exit du système
        xorl    %ebx,%ebx# mise à zéro du 1er paramètre en utilisant
                        # xor, c'est à dire ou exclusif
        int     $0x80     # appel de l'interruption 128 -> GNU/Linux
        ret             # fin du programme et retour au système
```

4. Sauvez le fichier et quittez l'éditeur.

5. Assemblez le fichier en tapant « gcc btlm.s -o btlm »

6. Exécutez le programme « ./btlm »

7. Sauf problèmes liés à votre configuration logicielle, vous devez voir apparaître dans votre terminal la phrase « Bonjour tout le monde! »

8. Vous pouvez aussi exécuter votre programme au pas-à-pas en utilisant un débogueur, par exemple en tapant « insight ./btlm ». Le déboguer vous permet aussi d'observer au pas-à-pas l'évolution des registres, de la mémoire, etc.

16.2 Programmes corrigés

En prenant exemple sur l'exercice précédent, essayer de programmer sans modèle les programmes des travaux dirigés. Vous trouverez quelques corrections ci-dessous.

16.2.1 Correction de l'exercice 14.7 :

```
        .text           # directive de création d'une zone d'instructions
        .globl main     # directive de création d'une étiquette
                        # de portée globale
main:   movq    $0x89,  %rax    # copie des 64 bits de 0x89 dans %rax
        movq    $0x67,  %rbx    # copie des 64 bits de 0x67 dans %rbx
        pushq   %rax            # empile 0x89
        pushq   %rbx            # empile 0x67
        pushq   $0x45           # empile 0x45
        popq    %rax            # dépile et copie 0x45 dans %rax
        popq    %rax            # dépile et copie 0x67 dans %rax
        popq    %rbx            # dépile et copie 0x89 dans %rax
fin:    ret
```

16.2.2 Correction de l'exercice 15.8 :

```
                .data
Stock :         .byte            5      # nombre (8 bits) d'objets
                                        # dans la liste
                .long   0x12345678      # taille (32 bits) du premier
                                        # objet
                .word   0x1234          # poids  (16 bits) du premier
                                        # objet
        .word       0x5678      # volume (16 bits) du premier objet
        .long   0x87654321      # taille (32 bits) du deuxième objet
        .word       0x0123      # poids  (16 bits) du deuxième objet
        .word        0x4321     # volume (16 bits) du deuxième objet
```

```
        .long   0x01234567      # taille (32 bits) du troisième objet
        .word       0x1123      # poids  (16 bits) du troisième objet
        .word       0x2345      # volume (16 bits) du troisième objet
        .long   0x98765432      # taille (32 bits) du quatrième objet
        .word       0x0012      # poids  (16 bits) du quatrième objet
        .word       0x7654      # volume (16 bits) du quatrième objet
        .long   0x09876543      # taille (32 bits) du cinquième objet
        .word       0x1234      # poids  (16 bits) du cinquième objet
        .word       0x8765      # volume (16 bits) du cinquième objet


 .text                          # directive de création d'une zone
                                # d'instructions
 .globl main                    # directive de création d'une étiquette
                                # de portée globale

###################### Programme Principal ####################

main:           pushq $Stock            # empile l'adresse du tableau
                                        # d'objets
                call TriBulle           # appel de la procédure tribulle
fin:            retq                    # retour au système
                                        # d'exploitation

###################### Procedure Tribulle  ######################

TriBulle:
                movq    8(%rsp), %rsi   # %rsi pointe sur le tableau à
                                        # trier
                movq    $0, %rdx        # pour mettre à zéro la
                                        # partie haute de %rdx
                movb     (%rsi), %dl    # %rdx contient le nombre
                                        # d'objets à trier (N)
                subq    $1, %rdx        # %rdx est la limite de la
                                        # boucle Pour (N-1)
Repeter:
                call    FaitPermutations # appel de la procédure
                cmpb    $0, %bl         # %bl = 0 si pas de nouvelle
permutation
                jne     Repeter         # tant que permut = VRAI
                retq    $8              # retour à l'appel avec
                                        # dépilement du paramètre d'appel

########################   Procedure FaitPermutations
######################

FaitPermutations:
 movb    $0,     %bl                    # permut = FAUX
PourInit:       movq    $0,     %rcx                    # Pour i=0
PourTest:       cmpq    %rdx,   %rcx            # Jusqu'à n-1
                jae     FinPour                 # on sort de la boucle
                                                # quand i>=n-1
Si:             movw    5(%rsi, %rcx, 8), %ax   # compare T[i].poids
                cmpw    %ax, 13(%rsi, %rcx, 8)  # avec T[i+1].poids
```

```
                jae     FinSi
                call    Permute
                movb    $1, %bl                 # permut = VRAI
FinSi:          addq    $1, %rcx                # élément suivant
                jmp PourTest                    # Boucle Pour
FinPour: retq
############################   Procedure Permute
############################

Permute: pushq  %rdx                            # sauve le nombre
d'objets à trier
                movq    1(%rsi, %rcx, 8), %rax  # sauve T[i]
                movq    9(%rsi, %rcx, 8), %rdx  # sauve T[i+1]
                movq    %rdx, 1(%rsi, %rcx, 8)  # copie T[i+1] dans T[i]
                movq    %rax, 9(%rsi, %rcx, 8)  # copie ancien T[i] dans
                                                # T[i+1]
                popq    %rdx                    # restaure le nombre
                                                # d'objets à trier
                retq
```

16.2.3 Correction de l'exercice 15.9 :

Dans le but de lire au clavier et d'afficher à l'écran des nombres hexadécimaux, il est nécessaire de pouvoir passer d'une valeur numérique contenue dans une case mémoire à une chaîne de caractères et vice-versa. Une chaîne de caractères est une suite composée des codes ASCII (8 bits) de chacun de ses caractères, terminée par le code ASCII « 0 ». On sait que les codes ASCII des chiffres ('0' à '9') sont consécutifs et inférieurs aux codes ASCII des lettres ('a' à 'z '), également consécutifs :

car.	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'	.....	'a'	'b'	'c'	'd'	'e'	'f'
ASCII	48	49	50	51	52	53	54	55	56	57	.....	97	98	99	100	101	102
valeur	0	1	2	3	4	5	6	7	8	9		10	11	12	13	14	15

Il est donc facile de convertir un seul caractère compris entre '0' et '9' ou entre 'a' et 'f' en valeur numérique comprise entre 0 et 15 : si le code ASCII du chiffre hexadécimal est supérieur ou égal au code ASCII de 'a', il suffit de lui soustraire le code ASCII de 'a' pour obtenir une valeur comprise entre 0 et 5 et de lui ajouter 10 pour obtenir une valeur comprise entre 10 et 15. Si le code ASCII du chiffre hexadécimal est inférieur au code ASCII de 'a', il s'agit donc d'un caractère compris entre '0' et '9'. Il suffit alors de lui soustraire le code ASCII de '0' pour obtenir une valeur comprise entre 0 et 9.

On écrira donc la conversion « caractère -> valeur » de cette façon :

```
cmpb  $'a', %dl  # %dl contient le caractère à convertir
jb    CarChiffre # code ASCII < 'a' : c'est un chiffre
subb  $'a', %dl  # c'est une lettre
addb  $0xA, %dl  # conversion en entier de 4 bits
jmp Suite
                 # caractère suivant
CarChiffre:

subb  $'0', %dl

Suite:           # %dl contient le chiffre hexadécimal
```

À l'inverse, la conversion « valeur -> caractère » s'écrira :

```
cmpb  $0xA, %dl    # %dl contient le chiffre à convertir
      jb    ValeurChiffre # valeur < 10
      subb  $0xA, %dl     # c'est une lettre
      addb  $'a', %dl     # %dl contient son code ASCII
      jmp   Suite          # chiffre suivant
ValeurChiffre:
      addb  $'0', %dl
Suite:       # %dl contient le caractère hexadécimal
```

Ce sous-problème étant réglé, il nous reste à écrire le code pour traiter tous les chiffres/caractères du nombre. Dans le cas d'une conversion « valeur -> chaîne de caractères », il nous trouver un moyen d'extraire chacun des chiffres de la valeur. Dans le cas d'une conversion « chaîne de caractères -> valeur », il nous faut trouver un moyen d'assembler chacun des chiffres de la valeur. Pour l'extraction des chiffres d'une valeur, on peut procéder par divisions par 16 successives, le reste de la division donnant le chiffre hexadécimal de poids faible. On peut réaliser la même chose avec des décalages à droite de 4 bits. Pour l'assemblage des chiffres d'une valeur, on peut procéder par multiplication par 16 par décalages à gauche de 4 bits.

```
        .data            # directive de création d'une zone de donnée
Nombre:
        .quad   0        # un nombre de 64 bits
Base:   .quad   16       # la base 16 (dite "hexadécimale")
NombreASCII:             # adresse symbolique pointant sur le nombre
```

```
      .string "XXXXXXXXXXXXXXXX\n"        # en caractères ASCII
        .text           # directive de création d'une zone d'instructions
        .globl main     # directive de création d'une étiquette
                        # de portée globale
main:                   # main est l'adresse de début du programme

        pushq   $NombreASCII     # empile l'adresse du nombre sous
                                 #forme de chaîne ASCII
        call LitChaine           # Appelle la procédure de lecture
        pushq   $Nombre          # empile l'adresse du nombre à obtenir
        pushq   $NombreASCII     # empile l'adresse de la chaîne à
convertir
        call HexaVersEntierMul   # appelle la procédure de conversion par
                                 # multiplications
        addq    $0x11111111, Nombre      # Fait une addition
        pushq   Nombre                   # empile le nombre
        pushq   $NombreASCII             # empile l'adresse de la chaîne
                                         # à écrire
        call    EntierVersHexaDivise     # appelle la procédure
                                         # de conversion par divisions
        pushq   $NombreASCII             # empile l'adresse du nombre
                                         # sous forme de chaîne ASCII
        pushq   $17                      # empile le nombre de caractères
                                         # à afficher
        call AfficheChaine       # Appelle la procédure d'affichage

        pushq   $NombreASCII     # empile l'adresse du nombre sous forme
                                 # de chaîne ASCII
        call LitChaine           # Appelle la procédure de lecture
        pushq   $Nombre          # empile l'adresse du nombre à obtenir
        pushq   $NombreASCII     # empile l'adresse de la chaîne à
convertir
        call HexaVersEntierDecale# appelle la procédure de conversion
                                 # par décalages

        subq    $0x111111, Nombre# Fait une soustraction

        pushq   Nombre           # empile le nombre
        pushq   $NombreASCII     # empile l'adresse de la chaîne à écrire
        call    EntierVersHexaDecale     # appelle la procédure de
                                         # conversion par décalages
        pushq   $NombreASCII             # empile l'adresse du nombre
                                         #sous forme de chaîne ASCII
        pushq   $17      # empile le nombre de caractères à afficher
        call AfficheChaine       # Appelle la procédure d'affichage

      movl    $1,%eax    # sélection de la fonction exit du système
      xorl    %ebx,%ebx  # mise à zéro du 1er paramètre en utilisant
                         # xor, c'est à dire ou exclusif
      int     $0x80      # appel de l'interruption 128 -> GNU/Linux
fin:    ret

###############################################################################
############################
```

```
LitChaine :
        movl    $3,%eax  # sélection de la fonction read du système
        movl    $2,%ebx  # dernier paramètre de write : stdin
        movl    8(%rsp),%ecx      # premier paramètre de write :
                                  # l'adresse de la chaîne de caractères
                                  # à lire
        int     $0x80     # appel de l'interruption 128 -> GNU/Linux
        ret      $16      # retourne en dépilant le paramètre

################################################################################
############################

AfficheChaine :
        movl    $4,%eax  # sélection de la fonction write du système
        movl    $1,%ebx  # dernier paramètre de write : stdout
        movl    16(%rsp),%ecx     # premier paramètre de write : l'adresse
                                  # de la chaîne de caractères à afficher
        movl    8(%rsp),%edx      # le nombre de caractères à afficher
        int     $0x80             # appel de l'interruption
                                  # 128 -> GNU/Linux
        ret      $16              # retourne en dépilant le paramètre

################################################################################
############################

HexaVersEntierDecale:
        movq     8(%rsp), %rsi   # %rsi pointe sur la chaine de caractères
        movq    16(%rsp), %rdi   # %rdi pointe sur nombre issu de la
                                 # conversion
        movq    $0, %rcx         # position du caractère courant dans la
                                 # chaîne
        movq    $0, %rdx         # mets à zéro les bits de %rdx pour
                                 # opération CopieCarDecale
ConversionCarDecale :
        movb    (%rsi, %rcx), %dl        # récupération du caractère
                                         # courant dans %dl
        cmpb    $'a', %dl                        #
        jb      CarChiffreDecale         # code ASCII < 'A' :
                                         # c'est un chiffre
        subb    $'a', %dl                # c'est une lettre
        addb    $0xA, %dl                # conversion en entier de 4 bits
        jmp CopieCarDecale               # saut vers l'insertion dans
                                         # %rax
CarChiffreDecale:
        subb    $'0', %dl        # %dl contient le chiffre
CopieCarDecale:
        shlq    $4, (%rdi)       # décale %rax d'un chiffre (4 bits) vers
                                 # la droite
        orb     %dl, (%rdi)      # copie %dl dans les 4 bits de poids
                                 # faibles de %rax
        addq    $1, %rcx         # chiffre suivant
        cmpq    $15, %rcx
        jbe     ConversionCarDecale      # conversion terminée l
```

```
                                        # orsque %rcx est négatif
        ret $16                         # retourne en enlevant les
                                        # 2 paramètres de la pile
################################################################################
####

HexaVersEntierMul:
        movq     8(%rsp), %rsi   # %rsi pointe sur la chaine de caractères
        movq    16(%rsp), %rdi   # %rdi pointe sur nombre issu de la
                                 # conversion
        movq    $0, %rcx         # position du caractère courant dans la
                                 # chaîne
        movq    $0, %rbx         # mets à zéro les bits de %rbx pour
                                 # opération CopieCarDecale
        movq    (%rdi), %rax     # Copie le nombre courant dans %rax
ConversionCarMul:
        movb    (%rsi, %rcx), %bl        # récupération du caractère
                                         # courant dans %bl
        cmpb    $'a', %bl                        #
        jb      CarChiffreMul            # code ASCII < 'A' : c'est un
                                         # chiffre
        subb    $'a', %bl                # c'est une lettre
        addb    $0xA, %bl                # conversion en entier de 4 bits
        jmp CopieCarMul                  # saut vers l'insertion dans %rax
CarChiffreMul:
        subb    $'0', %bl                # %bl contient le chiffre
CopieCarMul:
        mulq    Base                     # multiplie le nombre courant
                                         # par la base :
                                         # résultat dans %rax
        addb    %bl,    %al              # ajoute le chiffre courant dans
                                         # les 4 bits de poids faible
        addq    $1, %rcx                 # chiffre suivant
        cmpq    $15, %rcx
        jbe     ConversionCarMul         # conversion terminée lorsque
                                         # %rcx est négatif
        movq    %rax, (%rdi)             # copie le nombre obtenu à
                                         # l'adresse demandée
        ret $16                          # retourne en enlevant les 2
                                         # paramètres de la pile

################################################################################
####

EntierVersHexaDivise:
        movq     8(%rsp), %rdi   # %rdi pointe sur la chaine de caractères
        movq    16(%rsp), %rax   # %eax contient le nombre à convertir
        movq    $15, %rcx        # position du caractère courant dans la
                                 # chaîne
ConversionChiffreDivise:
        movq    $0, %rdx
        divq    Base             # divise %rdx:%rax par 16 : le reste de
                                 # la division %rdx
                                 # est le chiffre de poids faible
```

```
                                # du nombre
                                # à convertir et
                                # le quotient (%rax) contient les
                                # autres chiffres à convertir
        cmpb    $0xA, %dl
        jb      ValeurChiffreDivise       # valeur < 10
        subb    $0xA, %dl                 # c'est une lettre
        addb    $'a', %dl                 # %dl contient son code ASCII
        jmp     CopieDivise               # saut vers la copie
ValeurChiffreDivise:
        addb    $'0', %dl       # %dl contient le code ASCII du chiffre
CopieDivise:
        movb    %dl, (%rdi,%rcx) # copie dans la chaine à la position %rcx
        subq    $1, %rcx        # chiffre suivant
        jns     ConversionChiffreDivise
                                # conversion terminée lorsque %rcx est
                                # négatif
        ret $16          # retourne en enlevant les 2 paramètres de
                         # la pile

##############################################################################
###

EntierVersHexaDecale:
        movq     8(%rsp), %rdi   # %rdi pointe sur la chaine de caractères
        movq    16(%rsp), %rax   # %eax contient le nombre à convertir
        movq    $15, %rcx        # position du caractère courant dans la
                                 # chaîne
ConversionChiffreDecale :
        movq    %rax, %rdx       # récupération du chiffre courant dans
%dl
        andb    $0xF, %dl        # masque les 4 bits de poids fort de %dl
        shrq    $4, %rax         # décale %rax d'un chiffre (4 bits)
                                 # vers la droite
        cmpb    $0xA, %dl
        jb      ValeurChiffreDecale       # valeur < 10
        subb    $0xA, %dl        # c'est une lettre
        addb    $'a', %dl        # %dl contient son code ASCII
        jmp     CopieDecale      # saut vers la copie
ValeurChiffreDecale:
        addb    $'0', %dl        # %dl contient le code ASCII du chiffre
CopieDecale:
        movb    %dl, (%rdi,%rcx) # copie dans la chaine à la position %rcx
        subq    $1, %rcx                 # chiffre suivant
        jns     ConversionChiffreDecale  # conversion terminée lorsque
                                         # %rcx est négatif
        ret $16                          # retourne en enlevant les 2
                                         # paramètres de la pile
```

16.2.4 Correction du 15.10

```
        .data           # directive de création d'une zone de donnée
        .align 8
```

```
m:       .double 0.0      # m : mensualité
k:       .double 0.0      # K : capital emprunté
t:       .double 0.0      # t : taux annuel proportionnel
n:       .quad   0        # n : nombre de mensualités
NBMOISDANSANNEE:
         .quad   12
EntreeK:
         .string "Veuillez entrer le capital emprunté\n"
EntreeT:
         .string "Veuillez entrer le taux annuel proportionnel\n"
EntreeN:
         .string "Veuillez entrer le nombre de mensualités\n"
AfficheM:
         .string "Le montant de la mensualité d'élève à %lf\n"
FormatEntreeDouble:
         .string "%lf"    # Pour les appels de scanf et printf (stdio.h)
FormatEntreeQuad:
         .string "%llu"   # llu : unsigned long long (non signés de 64 bits)

         .text            # directive de création d'une zone d'instructions
         .globl main      # directive de création d'une étiquette

                          # de portée globale
main:                     # main est l'adresse de début du programme
################ LECTURE DE K ######################
         subq     $8,%rsp                  # crée un paramètre obligatoire,
                                           # mais inutilisé pour printf
         movl     $EntreeK, %edi           # chaine de format
         movl     $1, %eax                 # stdout
         call     printf                   # affiche l'invitation à entrer k
         addq     $8, %rsp                 # libère le paramètre fictif
         movq     $k, %rsi                 # l'adresse du nombre à lire
         movq     $FormatEntreeDouble, %rdi         # chaine de format
         movq     $0, %rax                 # stdin
         call     scanf                    # Lit k
################ LECTURE DE T ######################
         subq     $8,%rsp                  # crée un paramètre obligatoire,
                                           # mais inutilisé pour printf
         movl     $EntreeT, %edi           # chaine de format
         movl     $1, %eax                 # stdout
         call     printf                   # affiche l'invitation à entrer t
         addq     $8, %rsp                 # libère le paramètre fictif
         movq     $t, %rsi                 # l'adresse du nombre à lire
         movq     $FormatEntreeDouble, %rdi         # chaine de format
         movq     $0, %rax                 # stdin
         call     scanf                    # Lit t
################ LECTURE DE N ######################
         subq     $8,%rsp                  # crée un paramètre obligatoire,
                                           # mais inutilisé pour printf
         movl     $EntreeN, %edi           # chaine de format
         movl     $1, %eax                 # stdout
         call     printf                   # affiche l'invitation à entrer n
         addq     $8, %rsp                 # libère le paramètre fictif
         movq     $n, %rsi                 # l'adresse du nombre à lire
         movq     $FormatEntreeQuad, %rdi  # chaine de format
```

```
        movq    $0, %rax                # stdin
        call    scanf                   # Lit n
################ Calcul de m #########################
        fldl    t                   # st(0) = t
        fildl   NBMOISDANSANNEE  # st(0)=12 ; st(1)=t
        fdivrp  %st(0),%st(1)    # st(0)=t/12
        fldl    k                # st(0)=K ; st(1) = t/12
        fmul    %st(1), %st(0)   # st(0)=K.t/12 ; st(1) = t/12
        fld1                     # st(0)=1.0 ; st(1)=K.t/12 ; st(2) = t/12
        fadd    %st(2), %st(0)   # st(0)=1.0+t/12 ; st(1)=K.t/12 ;
                                 # st(2) = t/12
        movq    n,       %rcx    # le nombre de mensualités
        fld1                     # st(0)=1.0 ; st(1)=1.0+t/12 ;
                                 # st(2)=K.t/12 ; st(3) = t/12
PuissanceN:
        fdiv    %st(1), %st(0)   # st(0)=(1.0+t/12)^-rcx ;st(1)=1.0+t/12 ;
                                 # st(2)=K.t/12 ; st(3) = t/12
        subq    $1,      %rcx    # rcx=rcx-1
        jnz     PuissanceN       # vers puissance suivante
        fld1                     # st(0)=1.0 ; st(1)=(1.0+t/12)^n ;
                                 # st(2)=1.0+t/12 ; st(3)=K.t/12 ;
                                 #st(4) = t/12
        fsubp   %st(0), %st(1)   # st(0)=1.0-(1.0+t/12)^n ;
                                 # st(1)=1.0+t/12 ; st(2)=K.t/12 ;
                                 # st(3) = t/12
        fdivrp  %st(0), %st(2)   # st(0)=1.0+t/12 ;
                                 # st(1)=(K.t/12)/(1.0-(1.0+t/12)^n) ;
                                 # st(2) = t/12
        fstp    %st(0)           # st(0)=(K.t/12)/(1.0-(1.0+t/12)^n) ;
                                 # st(1) = t/12
        fstpl   m                # stocke le résultat ; st(0)=t/12
        fstp    %st(0)           # vide la pile des registres flottants

################ Affiche le résultat #################
        pushq m                  # sauve m
        movsd   m,       %xmm0
        movl    $AfficheM, %edi  # chaine de format
        movl    $1, %eax         # stdout
        call    printf
        addq    $8, %rsp         # libère m

        movq    $1,%rax  # sélection de la fonction exit du système
        xorq    %rbx,%rbx        # mise à zéro du 1er paramètre en
utilisant
                         # xor, c'est à dire ou exclusif
        int     $0x80    # appel de l'interruption 128 -> GNU/Linux
fin:    ret
```

16.2.5 Calcul sur les nombres complexes (flottants) :

```
        .data
Nombre1:
        .double 12.34    # partie réelle : un nombre IEEE 754 de 64 bits
        .double 5.678    # partie imaginaire : un nombre IEEE 754 de
```

```
                        # 64 bits
Nombre2:
        .double  901.2    # partie réelle : un nombre IEEE 754 de 64 bits
        .double 0.345     # partie imaginaire : un nombre IEEE 754 de
                          # 64 bits
Nombre3 :
        .double  0.0      # partie réelle : un nombre IEEE 754 de 64 bits
        .double 0.0       # partie imaginaire : un nombre IEEE 754 de
                          # 64 bits


        .text             # directive de création d'une zone d'instructions
        .globl main       # directive de création d'une étiquette

                          # de portée globale
main:                     # main est l'adresse de début du programme

        pushq   $Nombre1
        pushq   $Nombre2
        pushq   $Nombre3
        callq   AdditionComplexes

        pushq   $Nombre1
        pushq   $Nombre2
        pushq   $Nombre3
        callq   SoustractionComplexes

        pushq   $Nombre1
        pushq   $Nombre2
        pushq   $Nombre3
        callq   MultiplicationComplexes

        pushq   $Nombre1
        pushq   $Nombre2
        pushq   $Nombre3
        callq   DivisionComplexes

       movl    $1,%eax   # sélection de la fonction exit du système
       xorl    %ebx,%ebx# mise à zéro du 1er paramètre en utilisant
                         # xor, c'est à dire ou exclusif
        int     $0x80     # appel de l'interruption 128 -> GNU/Linux
fin:    ret

#############################################################################
########################
AdditionComplexes :
        movq    24(%rsp), %rsi   # adresse du premier nombre : a+b.i
        movq    16(%rsp), %rdi   # adresse du second  nombre : c+d.i
        fldl     (%rsi)          # st(0)=a
        fldl     (%rdi)          # st(0)=c ; st(1)=a
        faddp   %st(0), %st(1)   # st(0)=a+c
        fldl    8(%rsi)          # st(0)=b ; st(1)=a+c
        fldl    8(%rdi)          # st(0)=d ; st(0)=b ; st(1)=a+c
        faddp   %st(0),  %st(1)  # st(0)=b+d ; st(1)=a+c
```

```
        movq    8(%rsp), %rdi    # %rdi pointe sur l'adresse du résultat
        fstpl   8(%rdi)          # dépile et stocke la partie
                                 # imaginaire (b+d)
        fstpl   (%rdi)           # dépile et stocke la partie réelle (a+c)
        ret $24                  # retour à l'appel avec dépilement des
                                 # 3 paramètres
#############################################################################
#######################
SoustractionComplexes :
        movq    24(%rsp), %rsi   # adresse du premier nombre : a+b.i
        movq    16(%rsp), %rdi   # adresse du second  nombre : c+d.i
        fldl     (%rsi)          # st(0)=a
        fldl     (%rdi)          # st(0)=c ; st(1)=a
        fsubrp  %st(0), %st(1)   # st(0)=a-c
        fldl    8(%rsi)          # st(0)=b ; st(1)=a-c
        fldl    8(%rdi)          # st(0)=d ; st(0)=b ; st(1)=a-c
        fsubrp  %st(0),  %st(1)  # st(0)=b-d ; st(1)=a-c
        movq    8(%rsp), %rdi    # %rdi pointe sur l'adresse du résultat
        fstpl   8(%rdi)          # dépile et stocke la partie imaginaire
                                 # (b-d)
        fstpl   (%rdi)           # dépile et stocke la partie réelle (a-c)
        ret $24                  # retour à l'appel avec dépilement des
                                 # 3 paramètres
#############################################################################
#########################
MultiplicationComplexes :
        movq    24(%rsp), %rsi   # adresse du premier nombre : a+b.i
        movq    16(%rsp), %rdi   # adresse du second  nombre : c+d.i
                                 # rappel : (a+bi)(c+di)=(ac-bd)+(ad+bc)i
        fldl     (%rsi)          # st(0)=a
        fldl     (%rdi)          # st(0)=c ; st(1)=a
        fmulp   %st(0), %st(1)   # st(0)=ac
        fldl     8(%rsi)         # st(0)=b ; st(1)=ac
        fldl     8(%rdi)         # st(0)=d ; st(1)=b ; st(2)=ac
        fmulp   %st(0), %st(1)   # st(0)=bd ; st(1)=ac
        fsubrp  %st(0), %st(1)   # st(0)=ac-bd

        fldl     (%rsi)          # st(0)=a ; st(1)=ac-bd
        fldl     8(%rdi)         # st(0)=d ; st(1)=a ; st(2)=ac-bd
        fmulp   %st(0), %st(1)   # st(0)=ad ; st(1)=ac-bd
        fldl     8(%rsi)         # st(0)=b ; st(1)=ad ; st(2)=ac-bd
        fldl     (%rdi)          # st(0)=c ; st(1)=b ; st(2)=ad ;
                                 # st(3)=ac-bd
        fmulp   %st(0), %st(1)   # st(0)=bc ; st(1)=ad ; st(2)=ac-bd
        faddp   %st(0), %st(1)   # st(0)=ad+bc ; st(1)=ac-bd


        movq    8(%rsp), %rdi    # %rdi pointe sur l'adresse du résultat
        fstpl   8(%rdi)          # dépile et stocke la partie imaginaire
                                 # (b-d)
        fstpl   (%rdi)           # dépile et stocke la partie réelle (a-c)
        ret $24                  # retour à l'appel avec dépilement des
                                 # 3 paramètres
#############################################################################
```

```
#########################
DivisionComplexes :
        movq    24(%rsp), %rsi   # adresse du premier nombre : a+b.i
        movq    16(%rsp), %rdi   # adresse du second  nombre : c+d.i
                                 # rappel : (a+bi)/(c+di)=
                                 # a(c-d)/(c²+d²) + b(d+c)/(c²+d²)
        fldl      (%rdi)         # st(0)=c
        fld     %st(0)           # st(0)=c ; st(1)=c
        fmulp   %st(0), %st(1)   # st(0)=c²
        fldl     8(%rdi)         # st(0)=d ; st(1)=c²
        fld     %st(0)           # st(0)=d ; st(1)=d ; st(2)=c²
        fmulp   %st(0), %st(1)   # st(0)=d² ; st(1)=c²
        faddp   %st(0), %st(1)   # st(0)=c²+d²
        fld     %st(0)           # st(0)=c²+d² ; st(1)=c²+d²
        fldl    8(%rdi)          # st(0)=d ; st(1)=c²+d² ; st(2)=c²+d²
        fldl     (%rdi)          # st(0)=c ; st(1)=d ; st(2)=c²+d² ;
                                 # st(3)=c²+d²
        fsubp   %st(0), %st(1)   # st(0)=c-d; st(1)=c²+d² ; st(2)=c²+d²
        fldl     (%rsi)          # st(0)=a ; st(1)=c-d ; st(2)=c²+d² ;
                                 # st(3)=c²+d²
        fmulp   %st(0), %st(1)   # st(0)=a(c-d) ; st(1)=c²+d² ;
                                 # st(2)=c²+d²
        fdivrp  %st(0), %st(1)   # st(0)=a(c-d)/(c²+d²); st(1)=c²+d²

        movq    8(%rsp), %rdx    # %rdx pointe sur l'adresse du résultat
        fstpl   (%rdx)           # dépile a(c-d)/(c²+d²) dans la partie
                                 # réelle du résultat
                                 # st(0)= c²+d²
        fldl    8(%rdi)          # st(0)=d ; st(1)=c²+d²
        fldl     (%rdi)          # st(0)=c ; st(1)=d ; st(2)=c²+d²
        faddp   %st(0), %st(1)   # st(0)=c+d; st(1)=c²+d²
        fldl    8(%rsi)          # st(0)=b ; st(1)=c+d; st(2)=c²+d²
        fmulp   %st(0), %st(1)   # st(0)=b(c+d) ; st(1)=  c²+d²
        fdivrp  %st(0), %st(1)   # st(0)=b(c+d)/(c²+d²)

        movq    8(%rsp), %rdi    # %rdi pointe sur l'adresse du résultat
        fstpl   8(%rdi)          # dépile b(c+d)/(c²+d²) dans la partie
                                 # imaginaire du résultat
        ret $24                  # retour à l'appel avec dépilement des
                                 #3 paramètres
#############################################################################
#########################
```

16.2.6 Correction du TD 15.11

```
        .data            # directive de création d'une zone de donnée
        .align 8

m1:     .word    0, 1, 2, 3
        .word    4, 5, 6, 7
        .word    8, 9, 10, 11
        .word   12, 13, 14, 15
```

```
m2:     .word    0, 4, 8, 12
        .word    1, 5, 9, 13
        .word    2, 6, 10, 14
        .word    3, 7, 11, 15

m3:     .long    0, 0, 0, 0
        .long    0, 0, 0, 0
        .long    0, 0, 0, 0
        .long    0, 0, 0, 0

FormatString16:                 # pour les appels à printf
        .string  " %.3d"
FormatString32:                 # pour les appels à printf
        .string  " %.6ld"
RetourChariot:
        .string "\n"
Annoncem1:
        .string "matrice m1:\n"
Annoncem2:
        .string "matrice m2:\n"
Annoncem22:
        .string "matrice m2 pivotée:\n"
Annoncem3:
        .string "m1 x m2 = \n"


        .text           # directive de création d'une zone d'instructions
        .globl main     # directive de création d'une étiquette
main:                   # main est l'adresse de début du programme
        pushq   $Annoncem1
        call    AfficheChaineSimple
        pushq   $m1
        call AfficheMatrice16    # Affiche la première matrice
        pushq   $Annoncem2
        call    AfficheChaineSimple
        pushq   $m2
        call AfficheMatrice16    # Affiche la deuxième matrice
        pushq   $m1
        pushq   $m2
        pushq   $m3
        call    ProduitScalaire
        pushq   $Annoncem22
        call    AfficheChaineSimple
        pushq   $m2
        call AfficheMatrice16    # Affiche la deuxième matrice après
                                 # inversion ligne/colonnes
        pushq   $Annoncem3
        call    AfficheChaineSimple
        pushq   $m3
        call AfficheMatrice32    # Affiche la matrice résultat

        movq    $1,%rax          # sélection de la fonction exit
                                 # du système
       xorq    %rbx,%rbx         # mise à zéro du 1er paramètre en
```

```
                                # utilisant xor, c'est à dire ou exclusif
        int     $0x80           # appel de l'interruption
                                # 128 -> GNU/Linux
fin:    ret

ProduitScalaire:
        movq    16(%rsp), %rdi  # adresse de la deuxième matrice
        pushq   %rdi            # pivote la 2e matrice pour avoir les
                                # vecteurs colonnes
        call    Pivote          # en lignes

        movq     8(%rsp), %rcx  # adresse de la matrice produit
        movq    16(%rsp), %rdi  # adresse de la deuxième matrice
        movq    24(%rsp), %rsi  # adresse de la première matrice
        movq    $0, %rax        # indice de la ligne de la première
                                # matrice
TraiteLigne:
        movq    $0, %rbx        # indice de la ligne de la transposée
                                # de la seconde matrice
TraiteColonne:
        movq    (%rsi, %rax, 8), %mm0    # charge la ligne de la 1ere
                                         # matrice dans mm0
        movq    (%rdi, %rbx, 8), %mm1    # charge la colonne de la 2e
                                         # matrice (ou ligne de sa
                                         # transposée) dans mm1
        pmullw  %mm1, %mm0      # %mm0 : partie basse du produit
                                # composante par composante
        call    SommeDesComposantes      # rdx contient la somme des
                                         # composantes 8 bits de %mm0

        movw    %dx, (%rcx)              # copie du résultat dans la
                                         # matrice produit
        movq    (%rsi, %rax, 8), %mm0    # charge la ligne de la 1ere
                                         # matrice dans mm0
        pmulhw  %mm1, %mm0               # %mm0 : partie haute du produit
                                         # composante par composante
        call    SommeDesComposantes      # rdx contient la somme des
                                # composantes 8 bits de %mm0
        shll    $16, %edx       # décale le résultat de 16 bits vers la
                                # gauche pour le mettre en partie haute

        addl    %edx, (%rcx)    # ajoute la partie haute du résultat dans
                                # la matrice produit
        addq    $4, %rcx        # pointe sur l'élément suivant à calculer
        addq    $1, %rbx        # indice de colonne suivant
        cmpq    $4, %rbx        # fin de colonne ?
        jne     TraiteColonne
        movq    $0, %rbx        # on recommence à la colonne 0
        addq    $1, %rax        # ligne suivante
        cmpq    $4, %rax        # dernière ligne ?
        jne     TraiteLigne
        ret $24

Pivote: movq    8(%rsp), %rsi   # adresse de la matrice à pivoter
```

```
        movq    $0,     %rax    # indice de ligne (l)
        movq    $0,     %rbx    # indice de colonne (c)
TraiteElement:
        cmpq    %rax,   %rbx
        jbe     ColSuivante     # élément suivant si l<=c
        movq    %rsi, %rdi      # on va échanger l'élément (l,c)
                                # avec l'élément (c,l)
        addq    %rax, %rdi
        addq    %rax, %rdi      # rdi:base + l * 2
        pushw   (%rdi, %rbx, 8) # élément (l,c) -> pile
        movq    %rsi, %rdx
        addq    %rbx, %rdx
        addq    %rbx, %rdx      # rdx:base + c*2
        pushw   (%rdx, %rax, 8) # élément (c,l) -> pile
        popw    %cx
        movw    %cx, (%rdi, %rbx, 8)# pile -> élément (l, c)
        popw    %cx
        movw    %cx, (%rdx, %rax, 8)# pile -> element (c ,l)
ColSuivante:
        addq    $1, %rbx
        cmpq    $4, %rbx
        jne     TraiteElement
        movq    $0, %rbx        # retour à la colonne 0
        addq    $1, %rax        # ligne suivante
        cmpq    $4, %rax
        jne     TraiteElement
        ret $8

SommeDesComposantes:
        pushq   %rax            # sauvegarde
        pushq   %rbx            # sauvegarde
        pushq   %rcx            # sauvegarde
        movq    $0,     %rdx    # la somme
        movq    %mm0,   %rax
        movb    $4,     %cl     # boucle de cl=4 jusqu'à 0 exclu

Somme:  movq    %rax,   %rbx
        andq    $0xFFFF, %rbx   # mot 16 bits de poids faible
        addq    %rbx,   %rdx    # ajoute à %rdx
        shrq    $16,    %rax    # décale %rax de 16 bits vers la droite
        subb    $1,     %cl     # compteur
        jne     Somme
        popq    %rcx
        popq    %rbx
        popq    %rax
        ret

AfficheMatrice16:
        movb    $4, %cl         # indice colonne
        movq    8(%rsp), %rsi   # pointeur vers la matrice
TraiteLigne16:
        movb    $4, %ch         # indice ligne
TraiteElement16:
        movq    $FormatString16, %rdi    # pointeur vers la chaîne de
```

```
                                        # formatage
        movq    $0, %rax        # stdout
        pushq   %rcx    # sauve rcx car modifié par la fonction printf

        pushq   %rsi    # sauve rsi car utilisé comme paramètre par la
                        # fonction printf
        movq    $0, %rbx# pour ne récuperer qu'un octet dans %rsi
        movw    (%rsi), %bx     # récupère le mot 16 bits à afficher
        movq    %rbx,   %rsi    # %rsi est le paramètre de donnée
                                # pour printf
        call    printf
        popq    %rsi
        popq    %rcx
        addq    $2, %rsi        # element suivant de la matrice
        subb    $1, %ch         # indice suivant
        jnz     TraiteElement16  # traite l'élément suivant dans la ligne
        movq    $RetourChariot, %rdi     # pointeur vers la chaîne saut
                                         # à la ligne
        movq    $0, %rax                 # stdout
        pushq   %rcx    # sauve rcx car modifié par la fonction printf

        pushq   %rsi    # sauve rsi car utilisé comme paramètre par la
                        # fonction printf
        call    printf  # affiche un saut à la ligne
        popq    %rsi
        popq    %rcx
        subb    $1, %cl  # ligne suivante
        jnz     TraiteLigne16
        movq    $RetourChariot, %rdi     # pointeur vers la chaîne
                                         # saut à la ligne
        movq    $0, %rax                 # stdout
        call    printf                   # affiche un saut à la ligne
        ret $8

AfficheMatrice32:
        movb    $4, %cl         # indice colonne
        movq    8(%rsp), %rsi   # pointeur vers la matrice
TraiteLigne32:
        movb    $4, %ch         # indice ligne
TraiteElement32:
        movq    $FormatString32, %rdi    # pointeur vers la chaîne
                                         # de formatage
        movq    $0, %rax                 # stdout
        pushq   %rcx                     # sauve rcx car modifié par la
                                         # fonction printf
        pushq   %rsi            # sauve rsi car utilisé comme
                                # paramètre par la fonction printf
        movq    $0, %rbx        # pour ne récuperer qu'un octet dans %rsi
        movl    (%rsi), %ebx    # récupère l'octet à afficher
        movq    %rbx,   %rsi    # %rsi est le paramètre de donnée pour
                                #  printf
        call    printf
        popq    %rsi
        popq    %rcx
```

```
        addq    $4, %rsi            # element suivant de la matrice
        subb    $1, %ch             # indice suivant
        jnz     TraiteElement32  # traite l'élément suivant dans la ligne
        movq    $RetourChariot, %rdi     # pointeur vers la chaîne saut à
                                 # la ligne
        movq    $0, %rax            # stdout
        pushq   %rcx                # sauve rcx car modifié
                                    # par la fonction printf
        pushq   %rsi                # sauve rsi car utilisé comme
                                    # paramètre par la fonction printf
        call    printf              # affiche un saut à la ligne
        popq    %rsi
        popq    %rcx
        subb    $1, %cl             # ligne suivante
        jnz     TraiteLigne32
        movq    $RetourChariot, %rdi     # pointeur vers la chaîne saut
                                         #à la ligne
        movq    $0, %rax                 # stdout
        call    printf                   # affiche un saut à la ligne
        ret $8

AfficheChaineSimple:
        movq    8(%rsp), %rdi
        movq    $0, %rax
        call    printf
        ret     $8
```